REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: National Science Foundation; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): 1800 G. St., NW, Washington, DC 20550	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

STATIC SCHEDULER FOR HARD REAL-TIME TASKS ON MULTIPROCESSOR SYSTEMS

12. PERSONAL AUTHOR(S): Tzu-Chiang Chang

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 01/92 TO: 09/92	1992, September 11	149

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): hard real-time systems; static scheduler; multiprocessor scheduling; earliest start first; earliest deadline first; simulated annealing;

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Task scheduling is one of the most important issues in a hard real-time system, because it is the schedule that ensures the tasks meet their deadlines and precedence constraints. Given a set of hard real-time tasks, to determine whether a feasible schedule exists such that the timing constraints and precedence constraints of the tasks are satisfied, and to produce such a schedule if one exists are the purposes of a static scheduler. The previous work done for the static scheduler in the computer aided prototyping system (CAPS) was mainly for the single processor environment.

The major work of this thesis is to develop several algorithms for scheduling hard real-time tasks on *multiprocessor systems* so that the associated timing and precedence constraints, as well as the communication requirements are met under the worst-case situation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Man-Tak Shing	22b. TELEPHONE (Include Area Code): (408) 646-2634; 22c. OFFICE SYMBOL: CS/Sh

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted; All other editions are obsolete

Approved for public release; distribution is unlimited

STATIC SCHEDULER FOR HARD REAL-TIME TASKS ON MULTIPROCESSOR SYSTEMS

by

Tzu-Chiang Chang
Capt. R.O.C. (Taiwan) Army
B.S. of Applied Math. in Computer Science
Chung Cheng Institute of Technology
the Republic Of China, 1986

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SYSTEM ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1992

ABSTRACT

Task scheduling is one of the most important issues in a hard real-time system, because it is the schedule that ensures the tasks meet their deadlines and precedence constraints. Given a set of hard real-time tasks, to determine whether a feasible schedule exists such that the timing constraints and precedence constraints of the tasks are satisfied, and to produce such a schedule if one exists are the purposes of a static scheduler. The previous work done for the static scheduler in the computer aided prototyping system (CAPS) was mainly for the single processor environment.

The major work of this thesis is to develop several algorithms for scheduling hard real-time tasks on *multiprocessor systems* so that the associated timing and precedence constraints, as well as the communication requirements are met under the worst-case situation.

TABLE OF CONTENTS

I. INTRODUCTION

A. BACKGROUND

This thesis addresses the development of hard real-time systems using the computer aided prototyping system (CAPS) and the prototype system description language (PSDL). The following sections introduce them in detail.

1. Hard Real-Time System

In (soft) real-time systems, tasks (activities, operations, jobs) are performed by the system as fast as possible, but they are not constrained to finish by a specific time. In a particularly restricted form of real-time systems, tasks have to be performed not only correctly, but also in a timely fashion; otherwise, there might be severe consequences. In other words, this kind of the real-time system is characterized by the fact that severe consequences will result if logical as well as timing correctness properties of the system are not satisfied. Such a real-time system is often referred to as a ***hard*** real-time system, as opposed to a ***soft*** real-time system. Therefore, a ***hard real-time system*** can be defined as a system in which the correctness of the system depends not only on the logical results of the computation, but also on the time at which the results are produced. If the results are not produced in a timely manner, disastrous consequences may occur. Most of the hard real-time systems are special-purpose and complex, requiring a high degree of fault tolerance, and are typically embedded in a larger system [LEV91].

Typically, a hard real-time system consists of a controlling system and a controlled system. For example, in an automated factory, the controlled system is the factory floor with its robots, assembling stations, and the assembled parts, while the controlling system includes the computer and human interfaces that manage and

coordinate the tasks on the factory floor. Thus, the controlled system can be viewed as the environment with which the computer interacts.

The controlling system interacts with its environment based on the information available about the environment, say, from various sensors attached to it. It is imperative that the state of the environment, as perceived by the controlling system, is consistent with the actual state of the environment. Otherwise, the effects of the controlling system's tasks may be disastrous. Hence, periodic monitoring of the environment as well as timely processing of the sensed information is necessary [STR88].

Timing correctness requirements arise in a hard real-time system because of the physical impact of the controlling systems' tasks upon its environment. For example, if the computer controlling a robot does not stop or turn it on time, the robot might collide with another object on the factory floor. Needless to say, such a mishap can result in a major catastrophe.

A number of new and sophisticated hard real-time applications are currently being contemplated by governments and industries around the world. In addition to automated factories, application can be found in avionics, undersea exploration, process control, flight control, robot and vision systems, as well as military applications such as C3I systems, strategic defense initiative (SDI) systems, and so forth.

In summary, the difference between the hard real-time system and the traditional system, where there is a separation between the correctness and performance, is that the correctness and performance are very tightly interrelated in the hard real-time system. Thus, hard real-time systems solve the problem of missing deadlines in ways specific to the requirements of the target application. However, it

should be said that the sooner a system determines that a deadline is going to be missed, the more flexibility it will have in dealing with the exception [STR88].

Depending on where the application is used, hard real-time systems can be categorized as either centralized or distributed systems.

• In ***centralized*** systems, the processors are located at a single point and the inter-processor communication cost is negligible compared to the processor execution cost. A multiprocessor system with shared memory or a system using single processor is the example of such systems.

• In ***distributed*** systems, the processors are distributed at different points and the inter-processor communication cost is not negligible compared to the processor execution cost. A local area computer network (LAN) is an example of such systems. The inter-processor communication cost is an important factor which must be explicitly taken into account in scheduling problems of distributed systems.

2. Computer Aided Prototyping System (CAPS)

The ***computer aided prototyping system*** (CAPS), currently being developed at the Naval Postgraduate School, is designed to improve software technology which helps the software engineers design hard real-time software systems, automatically construct a real-time schedule, and automatically generate an executable Ada prototype of the proposed system from the PSDL specification. The Ada prototype is a combination of CAPS-generated Ada programs and reusable atomic Ada components.

CAPS also supports system management and helps control a system's evolution [LUQ89]. This support helps designers give timely responses to modification requests and helps protect the system's integrity as it evolves,

extending its life. The CAPS consists of three primary subsystems: the user interface, the execution support, and the software database system (see Figure 1).

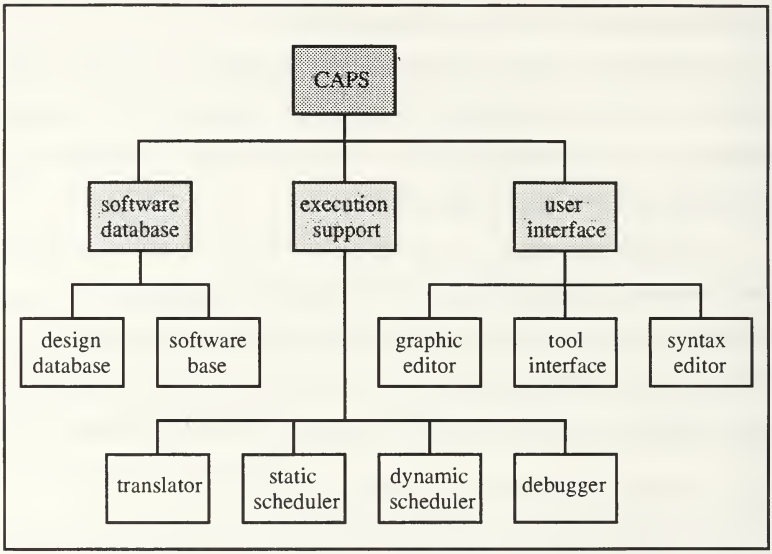


Figure 1: The Main Components and Their Associated Tools of CAPS

The user interface includes a graphic editor, a syntax-directed editor, and a tool interface. The graphic editor lets the designer edit a graphical representation of the prototype and automatically produces a PSDL representation that other CAPS tools can use. The designer can specify parts of a prototype using graphical objects to represent PSDL computational structures like tasks and data streams. The designer enters text annotations with the syntax-directed editor. The tool interface hides the details of the interfaces among CAPS tools from the designer.

The software database system, which includes a design database and software base, holds reusable components and manages the configuration. It uses existing object-oriented databases and formal models for prototyping design database and software base [BOE89].

The execution support system includes a translator, static scheduler, dynamic scheduler, and debugger. The translator generates codes that bind the reusable components extracted from the software database. Its main functions are to implement data streams, control constraints, and timers. The static scheduler uses several algorithms to allocate time slots for tasks with real-time constraints on single processor before execution begins [STR88]. If this allocation succeeds, all the tasks are guaranteed to meet their deadlines even in the worst case. The dynamic scheduler allocates time slots for tasks that are not time-critical. The debugger monitors timing constraints and various aspects of design integrity as the prototype runs, reports failures, and lets the designer adjust deadlines.

The CAPS is being developed as an ongoing research effort, and some of the functions just listed were not ready when the experiments were begun. Detailed information about CAPS is contained in [WHI89], [LUK88], [JAN88], and [MAR88].

3. Prototype System Description Language (PSDL)

The ***prototype system description language***, which integrates the tools in CAPS, provides the designer with a uniform conceptual framework and a high level system description and determines the properties of proposed designs via prototype execution and static analysis. It can describe prototypes of large software systems with hard real-time constraints on different levels of abstraction.

PSDL simplifies the design of a system with hard real-time constraints by presenting a high-level description in terms of networks of independent tasks to the designer, and automatically introducing any required interleaving of the codes via the execution support system.

PSDL has been designed to help the requirements analysts determine how to adjust the proposed functions of the system, the target architecture, or both to ensure that the requirements are feasible and provide the best service possible to the users of the proposed software. PSDL also provides a description of a proposed design that can be smoothly transformed into a final implementation after the requirements have been validated and the design has been verified.

B. SCHEDULING PROBLEM

Task scheduling is one of the most important issues in hard real-time systems because it is the schedule that ensures all tasks meet their deadlines. A scheduling problem in a hard real-time system is characterized by the nature of the tasks to be scheduled, the constraints associated with the system, and the usage of the schedule. Each of these is described in the proceeding sections.

1. Nature of the Task (Operator)

A ***task*** is a software module that can be invoked to perform a particular function and it is the entity of the scheduling problem in a system. In PSDL, a task corresponds to an operator and is represented by a vertex (circle) in the implementation graph. So, we will use "operator" as a synonym of "task" from now on. The following paragraphs introduce some characteristics of the task.

a. Time-Critical and Non-Time-Critical

A task is said to be ***time-critical***, sometimes it is called a hard real-time task, if there is at least one timing constraint associated with it, otherwise it is ***non-time-critical***. Ideally, the computer should execute time-critical tasks such that each task will meet its timing constraints, whereas it should execute the non-time-critical tasks such that the average response time of these tasks is minimized. The need to meet the timing requirements for all time-critical tasks is one issue that makes the scheduling problem a difficult one.

b. Periodic and Non-periodic

In hard real-time systems, tasks can be either periodic or non-periodic. *A **periodic*** task is defined as one which is activated exactly once per period P. In other words, once a periodic task is invoked at time t_0, then it will be activated exactly at time $(t_0 + P)$, $(t_0 + 2P)$, $(t_0 + 3P)$,....etc.

Non-periodic tasks are those whose activation are not periodic in nature. Such tasks can be subdivided into two categories [BUR91]: aperiodic and sporadic. The difference between these two categories lies in the nature of their activated frequencies. ***Aperiodic*** tasks are those whose activated frequency is unbounded. In the extreme, this could lead to an arbitrarily large number of simultaneously active tasks. ***Sporadic*** tasks are those who have a maximum frequency such that only one instance of a particular task can be active at a time.

c. Preemptive and Non-Preemptive

In hard real-time systems, tasks are also distinguished as preemptive and non-preemptive. A task is ***preemptive*** if its execution can be interrupted by other tasks at any time and its resumption can be at the same time on a different processor

or at a later time on any processor afterwards. A task is ***non-preemptive*** if it must finish without interruption once it starts. So far, non-preemptive periodic tasks have received little attention. To schedule a periodic task non-preemptively, it is usually assumed that the task is executed with a fixed time between successive executions of the same task on the same processor.

d. Dependent and Independent

A task is considered ***independent*** if there are no relationships between its execution and other tasks' executions. In other words, an independent task does not have to wait for execution until some other tasks finish their execution. Otherwise it is considered ***dependent***. Usually, a complex task, for example, one requiring access to many resources, is better handled by breaking it up into multiple sub-tasks and each requiring a subset of the resources. In PSDL, the sub-tasks are designated as ***atomic operators***. From now on, all the tasks to be scheduled are considered as atomic operators.

2. Constraints and Requirements

In a hard real-time system, a task is subject to the timing constraints, precedence constraints, communication requirements, as well as resource requirements. In this thesis, the resource requirements, except for the processor resources, are always met.

a. Precedence Constraints

The ***precedence constraints*** among a set of tasks specify the relations between the tasks. A precedence constraint ensures that a task (parent) which produces data for another task (child) will be scheduled to complete before data is required. These constraints can be specified between tasks scheduled on the same

processor as well as between tasks scheduled on different processors. Each precedence constraint is represented by an ordered pair of tasks. For example, a task T_i is said to precede task T_j, denoted as $T_i < T_j$, if T_i must finish before T_j begins. The whole precedence constraints between tasks being scheduled form a *precedence graph*. The precedence graph of a set of tasks is an acyclic directed graph. It may be a chain, a tree, a series-parallel graph, or an arbitrary one. In a static system, the precedence graph are known in advance. Figure 2 illustrates on example of the precedence graph in a system.

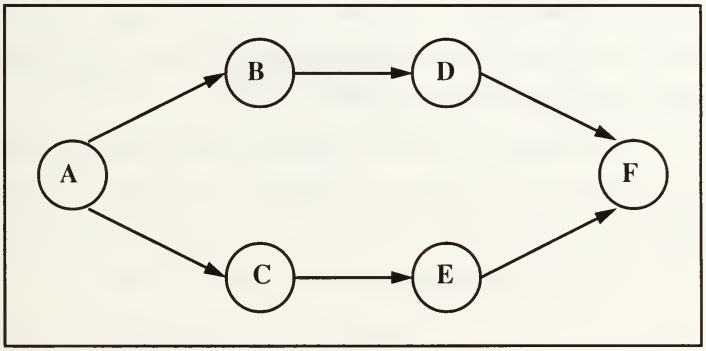


Figure 2: A Example of Precedence Graph

b. Timing Constraints

The timing constraints of a task are specified by giving bounds on one or more of the following basic timing parameters:

• The ***period*** (P): the time interval between any two consecutive temporal triggering events (instance) for a periodic task.

• The ***starting time*** (START): the actual time (instant) when a task starts to execute. It is also called "the firing time".

• The ***stop time*** (STOP): the time (instant) when a task finishes its execution. In PSDL, the time interval from the starting time to the stop time of a task forms a ***scheduling interval*** for the task.

• The ***arrival time*** (A): the time (instant) when a task is activated in the system. The arrival time for an instance of a periodic task with period P specifies the time at which the instance of the periodic task is activated. It is specified as follows:

$$A(i+1) = A(i) + P \quad \text{for } i \geq 1 \qquad \text{(Eq 1.1)}$$

where A(i) is the arrival time of the i-th instance of the periodic task.

• The ***ready time*** (R): The time (instant) when a task is ready to begin execution. It is the earliest possible starting time for a task. The ready time of a task is equal to or greater than its arrival time, i.e., $A \leq R$.

• The ***deadline*** (D): The time (instant) by which a task must finish. The deadline of instances of a periodic task with period P must satisfy the following inequality:

$$D(i) \leq A(i) + P \quad \text{for } i \geq 1 \qquad \text{(Eq 1.2)}$$

where D(i) is the deadline of the i-th instance of the periodic task.

Being a time-critical task, at least one of the following timing constraints associates with it:

(1) $R(i) \leq START(i)$

(2) $STOP(i) \leq D(i)$ for $i \geq 1$

where R(i), START (i) and STOP (i) are the ready time, the starting time and the stop time of the i-th instance of a periodic task respectively.

Figure 3 shows a graphic view of the timing parameters for a periodic task.

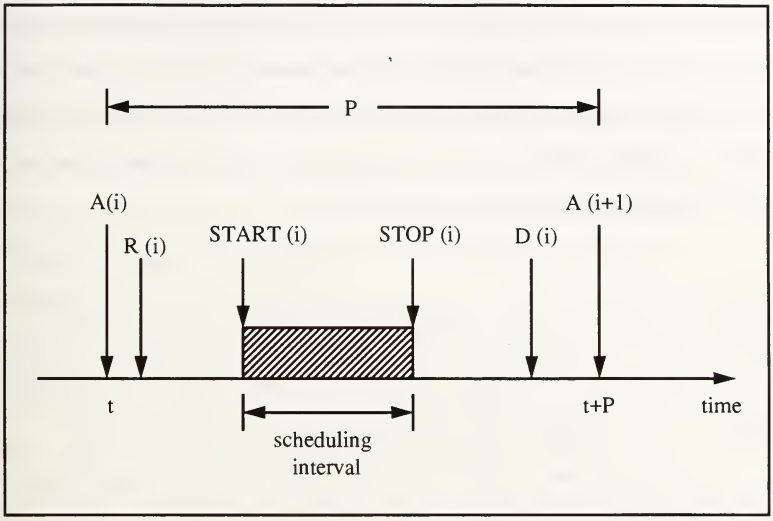


Figure 3: Timing Constraints for A Periodic Task

c. Communication Requirements

The communication requirement which is the time delay for the data transfer between tasks is constrained to those which have precedence relationship. In PSDL, it is specified by the "LINK" statement. The meaning of this constraint is that data can not be read from a stream until this time delay has elapsed.

3. The Usage of the Schedule

Depending on how and where the schedule is used, the scheduling problem can be categorized as follows.

a. Static & Dynamic Scheduling

A ***static*** approach schedules tasks off-line before the system begins to operate. Therefore, it requires the complete prior knowledge of the constraints and requirements of the system. Although this approach has low run-time cost, they are inflexible and hardly adapted to a changing environment or to an environment whose behavior is not completely predictable. Whenever there are new tasks which are added to such system, the schedule for the entire system must be recalculated and the cost in terms of time and money is very expensive.

In contrast, a ***dynamic*** approach progressively determines the schedule for tasks on-line and allows tasks to be dynamically activated. This approach, because of the way it is designed, involves higher run-time cost, but it is flexible and can be easily adapted to changes in the environment. Hence, it emerges as a challenging new problem especially for distributed hard real-time systems.

For a system using static scheduling approach, called a static system, the set of tasks along with their nature in the system is pre-specified, so the number of tasks to be scheduled and the associated constraints in the system is known beforehand. However, in a dynamic system, new tasks are allowed to arrive (to be activated) at unpredictable times so the number of tasks that must be scheduled changes at run time.

b. Uniprocessor & Multiprocessor Scheduling

A schedule can either assign tasks to single processor (uniprocessor) or multiprocessor system. In ***uniprocessor scheduling***, where tasks are scheduled only on a single time frame, each task can execute one by one based on the order of the schedule. There is no overlapping between any two execution of tasks, because the CPU can execute only one job at one time. On the other hand, ***multiprocessor***

scheduling assigns tasks to more than one processors, then different tasks can execute at the same time based upon the schedule on these processors. The scheduling problem on multiprocessor system is much more complicated than on single processor system. One important aspect of the multiprocessor is its application in real-time systems. Computer architecture had made rapid progress in the manufacture of chips. This makes processors cheaper than before. Progress is now limited by software problems. Scheduling problem is one of the problems that must be solved so that the utilization of the processors can achieve maximum.

C. OBJECTIVES

The static schedule for multiprocessor systems is produced for the execution of a prototype, which is a fixed number of sequences tasks being developed from the *prototype system description language* input specification for the prototype that obeys some predefined properties, such as timing constraints and precedence relationships. The number of sequences depends upon the number of processors which are available for scheduling. The static schedule gives the precise execution order and timing of tasks with hard real-time constraints in such a manner that all timing and precedence constraints are guaranteed to be met [OHE88].

This thesis builds upon work previously done in the development of the CAPS. The major work is to improve upon the existing version of the static scheduler for the hard real-time applications in single processor systems and to develop new algorithms for scheduling tasks on multiprocessor systems to satisfy the associated constraints of the problems.

The function of the scheduling algorithm is to determine, for a given fixed number of processors and a given set of tasks, whether a schedule (the sequence and

time periods) for executing the tasks such that the timing, precedence, and resource constraints of the tasks are met, and to produce such a schedule if one exists.

D. ORGANIZATION

The rest of this thesis is organized as follows:

Chapter II defines the basic terms necessary for the implementation of the static scheduler and surveys previous research about the scheduling problems. Chapter III describes the scope of the scheduling problem being concerned by CAPS and three scheduling algorithms being developed for the hard real-time systems. Chapter IV describes the system flow of the implementation for the static scheduler and explains the modification of the existing packages as well as the new packages being developed. Chapter V summarizes the whole work in this thesis and presents recommendations for future work. Appendix A gives some examples of the input test data for the new static scheduler. Appendix B lists the scheduled tables obtained from each scheduling algorithm for each case of the input test data. Appendix C and D are the modified ADA codes of the existing packages and the new ADA packages being developed for the static scheduler applied to multiprocessor systems.

II. SURVEY OF PREVIOUS WORK

Much research has been done in hard real-time scheduling problems. In this chapter, the previous research related to the static scheduling problem as well as the definition of terms used in the CAPS static scheduler will be discussed.

A. DEFINITION OF TERMS

The following paragraphs define the terms necessary to understand the static scheduling algorithms.

1. HBL

The Harmonic Block Length (HBL) is the least common multiple (LCM) of all the periods of tasks being scheduled. The reason to create the HBL is that once a schedule has been developed for a HBL, it can be repeated over and over again.

Scheduling periodic tasks naturally leads to periodic schedules. A schedule is called periodic with a period HBL if the following holds:

if task is executed at time t then it is also executed at time (t + HBL).

The algorithm to calculate the HBL is introduced in the package of "HARMONIC_BLOCK_BUILDER" [KIM89].

2. Instance

The ***instance*** of a periodic task is the repetition of this task in one Harmonic Block Length. The number of instances (N) for each task in a Harmonic Block Length is calculated as follows:

$$N := HBL / P \qquad \text{(Eq 2.1)}$$

3. Tardiness and Cost

The amount of time by which the stop time of an instance missed its deadline is called the ***tardiness*** of the instance. It is an essential quantity to calculate the cost of a schedule. If the stop time is less than or equal to the deadline, then the tardiness is zero. The tardiness of instance *i*, *T(i)*, of a task is calculated as follows:

$$T(i) := \max \{ 0, STOP(i) - D(i) \}, \quad \text{for } i \geq 1 \qquad \text{(Eq 2.2)}$$

The ***cost*** (C) of a schedule can be defined as either the largest tardiness or the sum of all tardiness experienced by tasks when executed according to the schedule, i.e.,

$$C := \max \{T(i) \mid \text{for all instances } i \text{ of all tasks in the schedules}\} \qquad \text{(Eq 2.3)}$$

or $$C := \sum T(i), \quad \text{for all instances } i \text{ of all tasks in the schedules} \qquad \text{(Eq 2.4)}$$

4. Latency

To express the behavior of distributed systems, PSDL has been extended to define optional latency attribute. The ***latency*** between two tasks is an upper bound on the duration of the time interval between the time instant a data value is written out to the data streams and the time instant that data value becomes available for reading by the next task. In the absence of explicit specifications, the latency of a stream has the default value zero (no delay). The purpose of these constraints is for the specification of communication constraints due to hardware limitations imposed by external constraints on how the software functions must be allocated to different physical nodes of a distributed system.

5. MET

The ***maximum execution time*** (MET) is the maximum time interval of execution for a task. It is the CPU time required to execute a task under worst-case

conditions. A correct implementation of a time-critical task must provide a guarantee of service with respect to bounded computational resources, i.e., the static scheduler must ensure that at least this much CPU time is allocated to a task between each activation time and its deadline.

6. Finish_Within

The ***finish_within*** is the hard deadline which is the largest permissible time interval between the temporal triggering event and the completion of the execution for a periodic task. The finish_within is constrained by the following relation:

$$P \geq finish\_within \geq MET, \qquad \text{and}$$

$$\text{finish\_within} := P, \qquad \text{by default.}$$

Hence, the deadline of the i-th instance of a time-critical periodic task equals:

$$D(i) := A(i) + \text{finish\_within}$$

$$\text{for } i \geq 1 \qquad \text{(Eq 2.5)}$$

7. Pipeline

If a task can *be* ***pipelined***, then different repetitions (instances) of the task can be scheduled on different processors with overlapping, i.e., more than one instance of this task can be firing at the same time. A time-critical task whose period is less than its maximum execution time, i.e., P < MET, can be realized only if it can be pipelined. If a periodic task can be pipelined then successive executions of the task can overlap in the schedule as long as its next instance of this task has been invoked.

8. Legal Solution and Feasible Solution

If a schedule satisfies all the precedence constraints, then it is said to be a ***legal*** solution.

A schedule is called a ***feasible*** solution, if it is legal and every task when executes according to the schedule meets its deadline, i.e., the schedule meets not only the precedence constrains but also the timing constraints.

9. Optimal Solution and Approximate Solution

A schedule is said to be ***optimal*** if, for any set of tasks, it always produces a schedule which satisfies all the constraints of the tasks.

A feasible solution with value (cost) close to the value (cost) of an optimal solution is called an ***approximate solution***. An approximate solution may not lead to the optimal result. However, there are many problems that have no exact solutions, and can only be solved by using approximation methods. We need an efficient heuristic algorithm to produce a close approximation to the optimal solution.

B. PREVIOUS RESEARCH

Many researchers have attempted to solve the general case of the scheduling problems with strong bounding conditions.The following paragraphs summarizes previous research relating to the static scheduling problems applied to both uniprocessor and multiprocessor environments for hard real-time systems.

1. Static Scheduling for Uniprocessor

Horn [HOR74] developed an optimal algorithm for scheduling preemptive independent tasks with arbitrary ready time and deadlines. His approach is based on the *earliest deadline first* policy. Liu and Layland [LIL73] developed a *rate-monotonic priority* scheme to determine the schedulability of a set of preemptive periodic tasks. This scheme assigns higher priorities to tasks with shorter periods. They showed that this scheme is optimal among fixed-priority schemes. Teixeira [TEI78] presented a fixed-priority assignment scheme for a slightly different

problem. He assumed that the relative deadline of a periodic task can be different from the period of the task. Sha and Lehoczky [LES86] described a technique to modify the periods of tasks in such a way that while tasks' timing constraints continue to be met, better processor utilization is achieved. This modification consists of breaking up one period task into two, each with half the computation time and half the period as the original task. The above approaches are different from the conventional priority-driven scheduling approaches, because they assign priorities to tasks based on a simple function of the timing constraints, instead of one that combines timing constraints and criticalness, of the periodic tasks.

Scheduling non-preemptive tasks is more difficult whether on uniprocessor or multiprocessor systems. Moore [MOO68] showed that the earliest deadline algorithm is optimal for scheduling a set of tasks with the same ready time. Kise [KIS78] developed an algorithm for the case in which a task has an earlier ready time if and only if it has an earlier deadline. Bratley, Florian, and Robillard [BFR71] developed an implicit enumeration algorithm to determine schedule for non-preemptive tasks with arbitrary ready time and deadlines [BFR75]. Baker and Su [BAS74] used a similar approach to minimize the maximum tardiness of tasks. Erschler et al [EFM83] developed a necessary condition for scheduling tasks with arbitrary ready time and deadlines. Their theories can be used to reduce the search space of an enumeration algorithm.

The original design of the static scheduler of CAPS was described in [LUQ86]. This design was further developed into a conceptual design for the pioneer prototype of the static scheduler [OHE88], and then was implemented by the Ada programming language in 1988 [JAN88],[MAR88].The current static scheduler consists of four algorithms for scheduling hard real-time tasks on single processor.

They are *earliest start first, earliest deadline first* [CER89], [KIM89], *exhaustive enumeration with branch and bound* [FAN90], and *simulated annealing* [LEV91].

2. Static Scheduling for multiprocessor

Horn [HOR74] described an algorithm to schedule preemptive independent tasks with arbitrary ready times and deadlines. His approach is based on the network flow method and considers processors with identical processing speed. Many researchers adopted a partition approach to solve the problems of scheduling preemptive periodic tasks on multiprocessor systems. The main idea of the approach is to partition a set of periodic tasks among a minimum number of processors such that each partition of the periodic tasks can be scheduled on one processor according to the earliest deadline scheme or the rate-monotonic priority scheme. Davari and Dhall [DAD86] showed that, if the earliest deadline scheme is used, a *bin-packing* algorithm can be used to determine a suboptimal partition pattern of periodic tasks among multiple processors preemptively. Bannister and Trivedi [BAT83] proposed a simple *best-fit partition* scheme. This scheme can be used in conjunction with both the earliest deadline scheme and the rate-monotonic priority scheme. For rate-monotonic priority scheme, Dhall and Liu [DHL78] improved these schemes and developed a more efficient *next-fit partition* scheme.

As described above, many of the scheduling algorithms designed for the preemptive periodic tasks are based on a fixed-priority assignment scheme. The advantage of a fixed-priority assignment scheme is that they have a very small scheduling overhead, because they are designed for prioritized-interrupt handling systems and the priority mechanism is often supported by hardware. However, in general, these schemes are very inflexible, because it is expensive to change the priority assignment once it is fixed on a system.

To schedule the non-preemptive tasks, a polynomial optimal algorithm is available only for the tasks with unit computation time [SIM80], [SIM83], [SIS84], [LAF76]. Otherwise, there is no polynomial optimal algorithm available so far for scheduling non-preemptive tasks on multiprocessor systems. Bratley, Florian, and Robillard [BFR75] developed a multi-stage enumeration algorithm to schedule tasks with arbitrary ready time and deadlines. Because the worst case exponential time complexity, the approach is designed to run off-line. Blazewicz, Drabowski, and Weglarz [BDW86] investigated an interesting scheduling problem in which tasks need multiple processors at the same time for processing. They showed that polynomial-time algorithms exist if the number of processors and the processing time required by tasks are constant.

If precedence constraints are subject to the tasks in multiprocessor system, the scheduling problems will be much more difficult. For example, scheduling tasks with arbitrary precedence constraints and unit computation time is *NP-hard* both for the preemptive and the nonpreemptive cases [ULL75], [ULL76].

Kasahara and Narita [KAN84] have developed an implicit heuristic search algorithm to determine the minimum schedule length for a set of nonpreemptive tasks with arbitrary precedence constraints. They showed that their enumeration algorithm can provide optimal or suboptimal solutions to large-scale problems within a time limit. However, the worst case execution time grows exponentially. Elsayed [ELS82] presented a number of heuristic algorithms for finding suboptimal solution to a similar scheduling problem. These heuristic algorithms do not enumerate over multiple paths in a search space. They are designed based on a straightforward topological search scheme and the *critical path* method combined with a heuristic rule. Therefore, such heuristic algorithms are much more efficient than the implicit enumeration algorithm described above. Hsu [HUS90] introduced some basic

concepts needed to schedule tasks in the multiprocessor system in CAPS. But there is no implemented codes available.

III. ALGORITHM DESIGN

This chapter describes the assumptions of the scheduling problem which is going to be dealt with, and then briefly describes three scheduling algorithms being developed for the hard real-time tasks in multiprocessor environment.

A. ASSUMPTIONS

Because this research is pioneers work in developing the static scheduler for CAPS in multiprocessor environments, some assumptions about the scheduling problem must be clarified before designing the algorithm.

(1) Since the purpose of the effort is to develop a static scheduler, all the requirements are static in nature (off-line schedule).

(2) Let T := $\{o_1,....,o_n\}$ be a set of n periodic tasks. For each $o \in T$, a maximum execution time MET(o) and a period P(o) are given. If the task is a sporadic task, it has to be converted into a periodic task. (periodic tasks)

(3) Once an execution of a task is started, it will be completed without interruption from the same processor. (non-preemptive tasks)

(4) The processors are supposed to be identical, that is to say, each task can be executed on any processor and the time to execute each task does not depend on the processor. Furthermore, a processor can only execute one task at a time. (identical processors)

(5) The ready time of the first instance for the operator who has no parent is assumed to be 0, but for the operator who has parents is the actual starting time of its first instance.

(6) In the scheduling process, two attributes are defined. The "LOWER" is the lower bound of the starting time for an instance of an operator, that is the ready time

for that instance, which ensures that at least one period is passed from the ready time of previous instance.

The "UPPER" is the upper bound of the starting time for an instance of an operator, which ensures that the instance is scheduled early enough so that it can finish execution prior to the deadline.

(7) If o_1 and $o_2$are periodic operators, then instance i_1 of o_1 precedes instance i_2 of o_2, written as $(o_1, i_1) < (o_2, i_2)$, if and only if o_1 is a parent of o_2 and $(i_1 - 1) * P(o_1) = (i_2 - 1) * P(o_2)$. The purpose of the second condition is to define corresponding synchronization points for the operators, as explained below.

If $(o_1, i_1) < (o_2, i_2)$ then instance i_1 of operator o_1 must complete execution before instance i_2 of operator o_2 can start execution, and instance i_2 of operator o_2 must read its inputs before instance $i_1 + 1$ of operator o_1. The purpose of this constraint is to allow the instance (o_2, i_2) to operate on the data produced by the instance (o_1, i_1). The first constraint is necessary to ensure the output of (o_1, i_1) has been produced before it is used by (o_2, i_2), and the second constraint is needed to ensure that the output of (o_1, i_1) is not over-written by the output of $(o_1, i_1 + 1)$ before it can be read by (o_2, i_2). When such a relation is guaranteed between two instances of periodic operators, the two instances are synchronized. The PSDL scheduling policy guarantees that the first instances of any two periodic operators connected by the precedence graph must be synchronized, and that subsequent instances are synchronized at intervals corresponding to the least common multiple of the periods of the two operators. In particular, if both operators have the same period, then they are synchronized for every pair of corresponding instances. This is illustrated in Figure 4 on next page.

period (a)	period (b)	synchronization points		
2	2	(a, 1) < (b, 1),	(a, 2) < (b, 2),	(a, 3) < (b, 3)....
2	3	(a, 1) < (b, 1),	(a, 4) < (b, 3),	(a, 7) < (b, 5)....
2	4	(a, 1) < (b, 1),	(a, 3) < (b, 2),	(a, 5) < (b, 3)....

Figure 4: Synchronization Points for Connected Periodic Operators

B. EARLIEST STARTING TIME FIRST ALGORITHM

The earliest starting time first algorithm assigns the highest priority to the task with the earliest starting time, i.e. the task with the least lower bound of the starting time among all activated tasks will be scheduled first. This algorithm schedules the task with the earliest starting time to execute on the earliest available processor to ensure all its descendants tasks can execute as soon as possible, so as not to violate the deadline constraints. The reason to choose the earliest available processor is to minimize the processor idle time.

In this algorithm, a priority queue is used for storing the information about the tasks which are going to be scheduled. The order of the priority queue depends on the lower bound of the starting time of each task (instance). The task, located at the top of the queue, with the smallest lower bound of the starting time will be extracted from the queue and be scheduled. Whenever a task is extracted from the priority queue, its lower bound has to be re-calculated. A recursive procedure is used to perform this function. Once there is a synchronized point between the task and its parent, the lower bound of the task has to be adjusted until all its parents with

synchronization has been completed. If any one of its parents with synchronization has not been scheduled, the task is pushed into a waiting list. The next task is then extracted from the top of the priority queue. The process keeps going until a task's lower bound has been completely calculated according to the stop time of all its parents having synchronization. The tasks in the waiting list are popped out and re-inserted into the priority queue. The earliest starting time first algorithm is described as follows.

```
BEGIN -- earliest starting time first algorithm
  while not empty (WORK_LIST) loop
     schedule the task;
     if INSTANCE < (HBL / P) then
       create an additional node for next instance of the same task;
       insert it into the priority queue;
     end if;
     insert all its children whose parents have all been scheduled
          into the priority queue;
     next (WORK_LIST);
  end loop;
  while not empty (priority queue) loop
     extract the task from the top of the priority queue;
     if INSTANCE = 1 then
       schedule the task;
       insert all its children whose parents have all been scheduled;
     else
       determine the new lower bound;
```

```
        schedule the task;
      end if;
      if START > UPPER or STOP > HBL then
        VALID_SCHEDULE := false;
      end if;
      if INSTANCE < (HBL / P) then
        create an additional node for next instance of the same task;
        insert it into the priority queue;
      end if;
    end loop;
    reverse the schedule;
  end EARLIEST_START;
```

-- The WORK_LIST links the tasks which have no tasks precede them.

-- The steps of scheduling a task include: assign a processor to the task, calculate the upper bound, the instance number, the starting time and the stop time for the given task.

-- according to the method for adding new tasks into the linked list, the schedule was developed in the reverse order, so it need to be reversed.

C. EARLIEST DEADLINE FIRST ALGORITHM

The earliest deadline first algorithm, similar to the earliest starting time first algorithm, assigns the highest priority to the task with the earliest deadline. In other words, this algorithm schedules the task with the smallest upper bound of starting

time to execute, before other tasks, on the earliest available processor to ensure that the task meets its deadline constraint. The reason to choose the earliest available processor is the same as mentioned in the earliest starting time first algorithm. The task with the smallest upper bound of the starting time is the most urgent task should execute before any other one because if the task was postponed for a moment, the deadline might be missed.

In this algorithm, a priority queue is also used for storing the information about the tasks which are going to be scheduled. The task located at the top of the queue has the smallest upper bound of starting time and will be extracted first. The algorithm, similar to the earliest starting time first algorithm, is described as follows.

```
BEGIN -- earliest deadline first algorithm
  while not empty (WORK_LIST) loop
     calculate the upper bound for the task;
     insert it into the priority queue;
     next (WORK_LIST);
  end loop;
  while not empty (priority queue) loop
     extract the task from the top of the priority queue;
     if INSTANCE = 1 then
       schedule the task;
       insert all its children whose parents have all been scheduled
           into the priority queue;
     else
       determine the new lower bound;
```

```
        schedule the task;
      end if;
      if START > UPPER or STOP > HBL then
        VALID_SCHEDULE := false;
      end if;
      if INSTANCE < (HBL / P) then
        create an additional node for next instance of the same task;
        insert it into the priority queue;
      end if;
    end loop;
    reverse the schedule;
  end EARLIEST_DEADLINE;
```

D. SIMULATED ANNEALING ALGORITHM

1. Generic Description

The use of simulated annealing to solve combinatorial optimization problems is an area that has received much attention lately. Combinatorial optimization problems are those whose configuration of elements are finite or countably infinite. An example of a combinatorial optimization problem is the assignment problem where there are numbers of personnel available to do an equal number of jobs. The cost for each person to do each job is known. The goal is to assign each person to a job so that the total cost is as small as possible [OTV89]. There is a wide range of combinatorial optimization problems that the simulated

annealing approach can be utilized. These include graph partitioning, graph coloring, number partitioning, VLSI design, and travelling salesman type problems.

Simulated annealing is based on the behavior of physical systems and the laws of thermodynamics. The way that liquids freeze and crystallize or metals cool and anneal are the principles upon which simulated annealing is based. At high temperature, liquid molecules move freely with respect to one another. As the liquid cools, this mobility is lost. Atoms line up and form a pure crystal that is at a minimum energy level. As the system cools slowly nature finds the minimum energy state [FLO84]. Examining simulated annealing in non-physical terms, a comparison is made to the concept of local optimization or iterative improvement. Local optimization repeatedly improves an initial solution until no further improvement of the solution is possible. This is known as iterative improvement or "hill climbing". Simulated annealing differs from local optimization in that random uphill movements (acceptance of a worse solution) are permitted. This prevents the algorithm from being trapped in a poor local optimal solution as demonstrated in Figure 5 on next page. Because simulated annealing avoids poor local optima, significantly better results can be found utilizing it as opposed to local optimization [JOH89].

The key to the use of the simulated annealing approach to solving combinatorial optimization problems is the random acceptance of worse iterative solutions. When the system is in a high energy state (high temperature), the probability is greater that a worse iterative solution is accepted. As the system cools this probability decreases, but even at the lower energy states the probability for making an uphill move still exists.

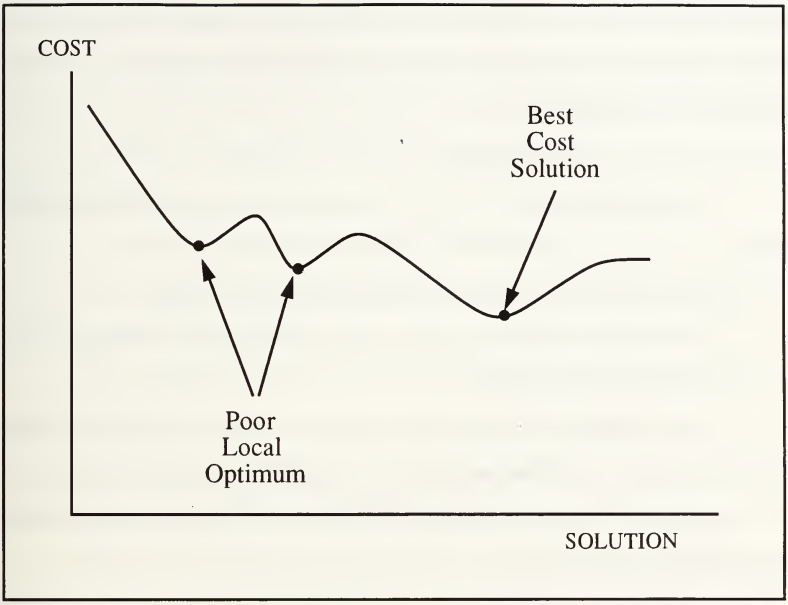


Figure 5: Cost/Solution Graph Demonstrating Local Minimum Solution

As indicated in Figure 5 above, uphill moves allow the algorithm to leave a poor local solution (point A or point B) and reach a better solution in the vicinity of point C. This general scheme of always taking a downhill step while occasionally taking an uphill step is known as the Metropolis algorithm, named after Metropolis, the scientist, who with his coworkers first investigated simulated annealing in 1953 [FLO84].

The choice of a probability function to determine if an uphill movement is allowed is an important consideration. At each step of the simulated annealing algorithm a new state is constructed based on the current state. This new state is

constructed by randomly displacing or adjusting a randomly selected element. If this new state has a lower cost than the current state, the new state is accepted as the current state. If the new state has a higher cost than the current state, the new state is accepted with probability:

$$\exp(-\Delta e/kT)$$

This probability function is known as the Boltzman probability distribution where:

Δe = difference in cost between new state and current state

k = Boltzman's constant of nature relating temperature to energy

T = Current Temperature

A characteristic of this probability function is that at very high temperatures every new state has almost the same chance of being accepted as the current state. At low temperatures, the states with a lower cost have a higher probability of being accepted than the current state.

Simulated annealing is simple to implement and can be applied to a variety of combinatorial optimization problems. To apply annealing, a description of the problem state space, a procedure or routine to adjust the state, and some function to determine the cost of the solution in the state space are all that is required [OTV89]. The next section will address these requirements as well as examine the algorithm for simulated annealing.

2. Algorithm Description

The concept of simulated annealing is closely related to local optimization. According to Johnson [JOH89], simulated annealing, which is a combinatorial optimization algorithm, can be specified by identifying a set of solutions together with a cost function that applies a value to each solution. There exists an optimum

solution which has the minimum cost possible. (NOTE: There may be more than one optimum solution). Starting with an arbitrary initial solution, the algorithm attempts to improve on the initial solution by performing incremental changes on that solution. A cost function is used to evaluate each iterative solution that is developed.

To utilize the simulated annealing algorithm, the following four elements must be provided:

(1) A description of possible system configurations.

(2) A generator of random changes in this configuration, these changes are "options" presented to the system.

(3) An objective function E (analog of energy) whose minimization is the goal of the procedure.

(4) A control parameter, T (analog of temperature) and an annealing schedule which tells how T is lowered and cooled.

The annealing schedule sets the number of random changes sampled for each temperature T and rate at which T decreases. The range of the annealing temperature and the value of the annealing schedule are normally established from trial and error experimentation [FLO84]. A pseudo code representation of the simulated annealing algorithm based on the algorithm proposed in [JOH89] follows:

Input:

(1) The solution space of the optimization problem.

(2) The control parameters for the annealing process, which include

(a) T_0 - the initial value of the control temperature T,

(b) Freeze - the final value of T,

(c) R - the reduction factor for T (typically $0.70 <= R <= 0.99$),

(d) L - the maximum number of attempted moves at any each value of T,

(e) L_s - the maximum number of accepted moves at any each value of T.

Output:

An optimal or near-optimal solution.

Algorithm

Begin

Current_Solution := some solution from the given solution space;

$T := T_0$;

Best_Solution := Current_Solution;

While (T > Freeze) do

begin

$N := 0$; /* tracks the number of moves attempted at the current value of T */

$N_s := 0$; /* tracks the number of moves accepted at the current value of T */

while (($N < L$) and ($N_s < L_s$)) do

begin

/* perturb current solution randomly to obtain a new legal solution */

New_Solution := **move** (*Current_Solution*);

$N := N + 1$;

ΔC := **cost** (*New_Solution*) - **cost** (*Current_Solution*);

if (($\Delta C \leq 0$) or (**random** () <= $e^{-\Delta C/T}$))

then begin

Current_Solution := *New_Solution*;

$N_s := N_s + 1$

if (**cost** (*Current_Solution*) < **cost** (*Best_Solution*))

then *Best_Solution* := *Current_Solution*;

```
        end;
    end;
    T := T×R
end;
Output: (Best_Solution);
End.
```

The choice of values for T_0, R, and L have a significant impact on the annealing schedule. The higher the initial temperature, the higher the cooling factor, and the larger the number of trials at each temperature result in more solutions being examined in order to find an optimum solution. The goal in choosing these parameters is to ensure that a sufficient, but not excessive, number of solutions are examined. These values are normally chosen arbitrarily and adjusted through experimentation.

IV. IMPLEMENTATION

Previous works on the static scheduler in CAPS developed by Janson [JAN88], Cervantes [HUS90], and Levine [LEV91] all applied to uniprocessor environment. The three algorithms developed in this thesis are used for scheduling hard real-time periodic tasks in multiprocessor environment. The ADA programming language is used to develop the new static scheduler packages as well as to modify the existing packages. This chapter describes the system flow of the static scheduler, also explains the modification for the existing packages and new packages being developed.

A. SYSTEM FLOW

The system flow diagram is given by Figure 6 as follows.

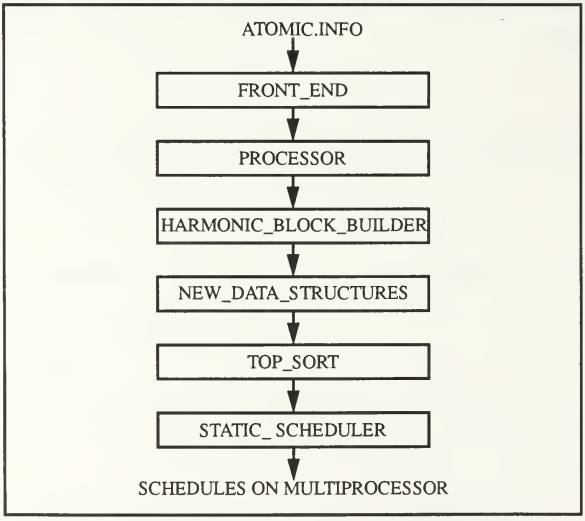


Figure 6: System Flow Diagram of the Static Scheduler

The first module "FRONT_END" reads the text input file "ATOMIC.INFO", which contains the operators' identifiers, timing information, and "LINK" statements which describe the PSDL implementation graphs, and separates the information of the time-critical operators and stores them in a linked list "OP_LIST". It also produces the number, "OP_COUNT", of the time-critical operators which are going to be scheduled.

The second module "PROCESSOR" calculates the periods for the sporadic operators and tests the time-critical operators' information from the linked list "OP_LIST". At this stage, all sporadic operators are converted to their periodic equivalents.

The third module "HARMONIC_BLOCK_BUILDER" uses to the period information of the periodic operators from the "OP_LIST" to determine the harmonic block length (HBL) of the static schedule as mentioned earlier.

The forth module "NEW_DAT_ STRUCTURES" is a generic package which is instantiated in the declarative part of the package "FRONT_END". It produces a *record* type of data structure named "GRAPH" which includes two entries, "OP_ARRAY" and "OP_MATRIX" (see Figure 7).

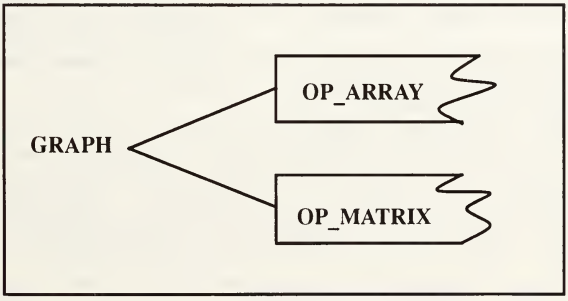


Figure 7: Graph Structure

• The first entry "OP_ARRAY" of "GRAPH" is a one-dimension array type which contains all relevant information about the operators to be scheduled from the "OP_LIST". Once the information is stored in the array, the operators can be referred to by their index number (position) in the array. This allows for immediate access of all relevant operator information instead of having to traverse a linked list to find the desired operator's information. Identifying operators by their index numbers as opposed to their names reduces the storage space required for operator's identifications throughout the static scheduler.

• The second entry "OP_MATRIX" of "GRAPH" is a two-dimension ($n \times n$) array type, where n is the number of operators to be scheduled, which contains the information about the parent-child relationships between operators as well as the pipeline information about each operator.

The fifth module "TOP_SORT" performs a topological sort on all the operators according to the information produced by the package of "NEW_DATA_STRUCTURE". It creates a true topological ordering which is not dependent on a specific ordering of the operators in the PSDL input file. The result, called "PRECEDENCE_LIST", is a linked list data structure containing each operator's index number according to the position of the operator in the "OP_ARRAY". In the head of the list, there is a dummy operator (index number 0) created to lead all other operators.

The last module "STATIC_SCHEDULER" is the heart of the whole system which combines the output of "TOP_SORT" (PRECEDENCE_LIST), "FRONT_END" (OP_COUNT), "NEW_DATA_STRUCTURES" (OP_ARRAY, OP_MATRIX), and "HARMONIC_BLOCK_BUILDER" (HBL) to produce a static schedule which can be applied to multiprocessor systems. This module includes three static scheduling algorithms which have been introduced in last chapter.

B. MODIFICATIONS ON EXISTING PACKAGES

This section describes the modifications of the existing packages in the proposed implementation.

1. DATA

The original "DATA" package contains the definitions of all types, instantiation of generic packages, and global variables used by the static scheduler for uniprocessor environment. Most of the packages remain the same. Some additional global variables and data types which are necessary for multiprocessor scheduling are added in this package.

- NOP : NATURAL := 4;

It is a global variable used for storing the number, which equals four in the experimental case, of the available processors which can be applied for the tasks' execution.

- type PROCESSOR_ARRAY is array (1..NOP) of NATURAL;

This data type is used for the variables, for example "PROCESSOR_STOP", which track the available time (instant) of each processor at each stage during the scheduling process.

- type SCHEDULE_ARRAY

  is array (1..NOP) of SCHEDULE_INPUTS_LIST.LIST;

It is used for the variables, for example "AGENDA", which store the pointers pointing to the linked lists of the schedule for each processor.

The other additional global variables are listed in "APPENDIX C".

2. NEW_DATA_STRUCTURES

Besides the original procedures and functions in the old version of this package, two functions are created. Also, one procedure and one data type are modified.

- type MATRIX_OP_INFO is

```
record
  PARENT          : INTEGER;
  CHILD           : INTEGER;
  DELAY_PIPELINE  : INTEGER;
end record;
```

This data type resides in the two dimension array (matrix) "OP_MATRIX" and the additional entry "DELAY_PIPELINE" is used to store the information about the operators whether they can be pipelined or not and the information about the latency between each pair of operators. If the "DELAY_PIPELINE" in the diagonal cell [*i*, *i*] of the matrix equals, then the operator *i* can be pipeline. The "DELAY_PIPELINE" in other cells [*i*, *j*], for $i \neq j$, of the matrix determines the latency between operator *i* and operator *j* (where *i*, *j* are the index number of the operators).

- procedure "PRODUCE_OP_MATRIX":

In the old version of this procedure, the "LINK" statements were skipped in the PSDL input file. Because the new version of the static scheduler considers the communication delay between operators, this procedure is modified in order to store the information of the latency between operators into the proper place.

• function "LATENCY":

This function is used for the new static scheduler to return the value of the latency between two operators. The first parameter is the parent's index number and the second parameter is the child's index number.

• function "PIPELINE":

This function returns the boolean constant "TRUE" if the operator can be pipelined, otherwise it returns "FALSE". Its parameter is the index number of the operator.

3. DIAGNOSTICS

This package includes four procedures: "OUTPUT_OP_ID", "OUTPUT_SCHEDULE", "OUTPUT_HARMONIC_BLOCK_LENGTH", and "OUTPUT_PRECEDENCE_LIST". These procedures output the information to the terminal for the purpose of diagnostics. The procedure "OUTPUT_SCHEDULE" is modified in order to output the schedule which will be applied to the multiprocessor other than the uniprocessor to the terminal.

C. NEW PACKAGES

There are two packages, "UTILITY_PKG" and "NEW_SCHEDULER_PKG", created in the new version of the static scheduler. They are the major effort of the implementation of the static scheduler. Each will be described in the following paragraphs.

1. UTILITY_PKG

This package consists of procedures and functions which help the scheduling algorithms to solve specific problems.

a. RANDOM_INITIALIZE

This procedure which was separate in the original version of the static scheduler initializes the random number generator. The input parameter should be an odd integer.

b. RANDOM_NEXT

This function creates a random number which has a type of "FLOAT" ranging from 0 to 1 and was originally separate in the old version.

c. DETERMINE_THE_UPPER

It is a procedure that calculates the upper bound of the starting time for each instance of the tasks. If the starting time of an instance exceeds its upper bound, the instance may miss its deadline. In other words, this task violates its timing constraints and the resulting schedule would not be feasible.

d. DETERMINE_START_STOP

This procedure calculates the actual starting time and stop time (scheduling interval) for an instance of a task. Even though an instance has a lower bound of the starting time, it still can not execute unless there is at least one processor available after the lower bound. If there are more than one processor available after this lower bound, the processor with the earliest release time will be chosen.

e. CREATE_ADDL_NODE

After an instance of an operator has been scheduled, the operator will be checked if it has more instances in the HBL to be scheduled. If so, an additional node of the scheduling information about the next instance of this operator will be created with a record data type of "SCHEDULE_INPUTS" declared in the "NEW_DATA_STRUCTURES" generic package. The entry "THE_START" of this

additional node is used to store the information about the synchronization instead of the starting time of next instance, because it is not known at this time when the node is created. This process includes in the "CREATE_ADDL_NODE" procedure.

f. TEST_SCHEDULE

This procedure calculates the cost of a given schedule. In other words, it finds out the maximum tardiness of all instances who miss their deadlines and whose stop time exceeds the HBL. If the cost equals 0, a feasible schedule is found. Otherwise, the given schedule is not a feasible solution.

g. ANNEAL_FUNCTION

This function used in the "*simulated annealing*" algorithm returns a "FLOAT" type number between 0 and 1. After comparing this number with the random number, the annealing process then decides whether to accept the new solution or not. The first parameter of this function is the cost of the new solution, and the second parameter is the cost of the original one. The last parameter is the current temperature.

h. ADJUST_PRECEDENCE

This procedure used in the "*simulated annealing*" algorithm adjusts the "PRECEDENCE_LIST" to get a new ordering of the operator.

2. NEW_SCHEDULER_PKG

In this package, three algorithms were developed:

- "EARLIEST_START"
- "EARLIEST_DEADLINE"
- "SIMULATED_ANNEALING"

Each of these has been introduced in chapter three. The following paragraphs describe the procedures in "SIMULATED_ANNEALING".

a. SCHEDULE_1st_INSTANCES

This procedure schedules the first instance for each operator. Its algorithm is described as follows.

```
BEGIN -- schedule the first instance for each operator
    duplicate P_LIST to WORK_LIST;
    while not empty (CHILD_LIST of the dummy node) loop
        calculate the upper bound, start time, stop time for the task;
        schedule the task; -- (lower bound := 0)
        remove this task from the WORK_LIST;
        if   INSTANCE = (HBL / P)      then
          remove this task from the P_LIST;
              -- no need to schedule the next instance for the task
        else
          create an additional node of the next instance;
          add it into an additional list (A_LIST); -- to be scheduled for the future
        end if;
        next (CHILD_LIST);
    end loop;
    while not empty (WORK_LIST)          loop
        calculate the lower bound, upper bound, start time and stop time;
        schedule the task;
        if   INSTANCE = (HBL / P)      then
          remove this task from the P_LIST;
```

```
            -- no need to schedule the next instance for the task
        else
          create an additional node of the next instance;
          add it into an additional list (A_LIST); -- to be scheduled for the future
        end if;
        next (WORK_LIST);
    end loop;
end SCHEDULE_1st_INSTANCES;

-- the P_LIST is the precedence list of the operators from the TOP_SORT
   package output.
```

b. SCHEDULE_REST_OF_BLOCK

It is a continuous work for the "SCHEDULE_1st_INSTANCE" procedure. This procedure schedules the rest of the instances other than the first instance of the tasks in one harmonic block length. After this procedure, an initial solution is obtained for the annealing process to anneal if needed. The algorithm is described as follows.

```
BEGIN -- schedule the rest instances other than the first instance in a HBL
    while not empty (P_LIST) loop
        duplicate P_LIST to WORK_LIST;
        while not empty (WORK_LIST)      loop
            extract the same task of WORK_LIST from the A_LIST;
            DETERMINE_THE_LOWER; -- recalculate the lower bound
            if the lower bound has been recalculated then
```

```
            schedule the task;
            if   INSTANCE = (HBL / P)      then
              remove this task from the P_LIST;
                -- no need to schedule the next instance for the task
            else
              create an additional node of the next instance;
              add it into an additional list (A_LIST);
                  -- to be scheduled for the future
            end if;
            remove this task from the A_LIST;
          end if;
          next (WORK_LIST);
      end loop;
  end loop;

end SCHEDULE_REST_OF_BLOCK;
```

-- The "P_LIST" is part of the PRECEDENCE_LIST coming from the output of the "SCHEDULE_1st_INSTANCES" procedure.

-- The sub-procedure"DETERMINE_THE_LOWER" checks all the parents of the task to be scheduled. Whenever there is any parent having a synchronized point with this task, the lower bound of the task has to be recalculated. If the parent has not been scheduled for the synchronized instance, the task can not be scheduled at this time. Then the sub-procedure returns a waiting message.

c. ANNEAL_PROCESS

Actually this procedure is derived from the generic simulated annealing algorithm introduced in chapter three. It occurs when the initial solution is not feasible. It uses a priority queue, named "QUE", as in the "earliest starting time first" algorithm. The order of the tasks in "QUE" depends on the lower bound of each task. The task with the smallest lower bound will be put on the top of the priority queue and will be extracted before any other tasks. The modification for the generic simulated annealing algorithm to solve the static scheduling problem is described as follows.

```
INPUT:
    HBL;
    AGENDA; --schedule on multiple processors.
    PENALTY_COST; -- cost for a given schedule.
    PRECEDENCE_LIST;
OUTPUT:
    AGENDA;
    PENALTY_COST;
    FEASIBLE_SOLUTION: boolean;

BEGIN -- annealing process
  duplicate AGENDA to BEST_AGENA and TEMP_AGENDA;
  initialize the temperature T;
  BEST_COST := PENALTY_COST;
  while not FEASIBLE_SOLUTION and T > Freeze loop
     clear ACCEPT_COUNT and TRIAL_COUNT;
     while not SOLUTION_FOUND and
```

```
  ACCEPT_COUNT < ACCEPT_NO and
  TRIAL_COUNT < TRIAL_NO              loop
REARRANGE_P := false;
find the first task whose START > UPPER on each processor;
insert them into (QUE);
while not empty (QUE)          loop
    extract the task from (QUE);
    recalculate the lower bound;
    if LOWER > UPPER then
      REARRANGE_P := true;
      exit the loop;
    end if;
    if LOWER < START then
      promote the task into a suitable time slot;
      if can not find a suitable time slot then
        REARRANGE_P := true;
        exit the loop;
      else
        remove the task from TEMP_AGENDA and QUE;
        next TEMP_AGENDA of the same processor;
        insert it into QUE;
      end if;
    else
      remove the task from QUE;
      next TEMP_AGENDA of the same processor;
      insert it into QUE;
```

```
    end if;
end loop;
if not REARRANGE_P then
  find the first task whose STOP > HBL on each processor;
  insert them into (QUE);
  while not empty (QUE)      loop
      extract the task from (QUE);
      OLD_LOWER : = LOWER;
      recalculate the lower bound;
      if LOWER = OLD_LOWER and LOWER = START then
        REARRANGE_P := true;
        exit the loop;
      else
        promote the task into a suitable time slot;
        if can not find a suitable time slot then
          REARRANGE_P := true;
          exit the loop;
        else
          remove the task from TEMP_AGENDA and QUE;
          next TEMP_AGENDA of the same processor;
          insert it into QUE;
        end if;
      end if;
  end loop;
end if;
if REARRANGE_P then
```

```
        ADJUST_PRECEDENCE;
        create another solution TEMP_AGENDA for the new precedence list;
      end if;
      TEST_SCHEDULE; -- output "TEMP_COST"
      if TEMP_COST < BEST_COST then
        BEST_COST := TEMP_COST;
        duplicate TEMP_AGENDA to BEST_AGENDA;
      end if;
      if TEMP_COST = 0 then
        FEASIBLE_SOLUTION := true;
      elsif REARRANGE_P or else
            TEMP_COST <= PENALTY_COST or else
            random number < ANNEAL_FUNCTION then
        ACCEPT_COUNT := ACCEPT_COUNT + 1;
        PENALTY_COST : =TEMP_COST;
        duplicate TEMP_AGENDA to AGENDA;
      else
        duplicate AGENDA to TEMP_AGENDA;
      end if;
      TRIAL_COUNT := TRIAL_COUNT + 1;
    end loop;
    T := T * COOL_FACTOR;
  end loop;
  AGENDA := BEST_AGENDA;
  PENALTY_COST := BEST_COST;
```

end ANNEAL_PROCESS;

-- The TEMP_AGENDA is a temporary schedule for the use of the annealing process. Its cost is called TEMP_COST.

-- REARRANGE_P is a boolean constant. Its value is "true" when the schedule can not be annealed any more and the precedence list needs to be adjusted.

-- A time slot is a time interval where the processor is free from any tasks. If a task's MET is less than or equal to this interval, the task can be scheduled during this interval.

V. CONCLUSIONS

A. RESULTS FROM THE STATIC SCHEDULER

Four examples of the input PSDL text files (atomic.info) are presented in appendix A in forms of tables and graphs. Each table lists the name (operator name), index number (No.), maximum execution time (MET), finish_within (Within), period (P) and the number of instances (N) in the HBL for each operator to be scheduled. The index number (No.) and the number of instances (N) are derived entries from the static scheduler. Each diagram in appendix A is a precedence graph for each case. Operators are represented by circles and their index numbers are shown in the circles. The latencies between operators are also shown in the diagrams by the numbers above the edges between circles.

The results from the static scheduler for each case of test data are shown in appendix B. The schedules on each processor are listed in separate tables. Each table lists the index number (OP), the instance number (IN), the starting time (START TIME), the stop time (STOP TIME), the lower bound (LOWER) and the upper bound (UPPER) of the starting time for each operator.

For case 1 and case 2 test data, the feasible schedules were found by all three algorithms. But in cases 3 and 4, feasible schedules can not be found when the *earliest deadline first* algorithm is used. The tasks which violate the timing constraints are highlighted in the table.

When using the *simulated annealing* algorithm, the initial solutions for case 1 and 2 are feasible. In other words, there is no need to use the annealing process to adjust the schedules. But in case 3 and 4, the initial solutions are not feasible. After the annealing process, the feasible schedules are found. Furthermore, during the

annealing process, case 3 needs to adjust the PRECEDENCE_LIST several times to obtain the feasible schedule. The proposed heuristic algorithm, based on the simulated annealing approach, appears to be the best compromise between simple-minded and exponential time algorithms implemented in CAPS.

B. SUMMARY

The thesis presents the research conducted to develop a static scheduler for hard real-time tasks in multiprocessor systems. Three scheduling algorithms were developed in the new static scheduler for CAPS. The *earliest starting time first* algorithm produces the schedule according to the earliest possible starting time of each instance of operators. The instance with the smallest earliest possible starting time (the smallest LOWER bound) will be scheduled before any others. In a similar way, the *earliest deadline first* algorithm produces the schedule according to the deadline of each instance of operators. The instance with the most urgent deadline, i.e. the smallest UPPER bound, will be scheduled before any others.

The *simulated annealing algorithm* first produces an initial solution based on the topological ordering (PRECEDENCE_LIST) of the operators. If the initial solution is feasible, there is no need to anneal the schedule, otherwise, the annealing process is used. The annealing process starts to find out the first instance, whose starting time is greater than its upper bound (i.e., missing its deadline) of the schedule on each processor. From these instances, the annealing process then starts to adjust each instance by choosing the instance with the smallest lower bound of the starting time (LOWER). During the annealing process, the ordering (precedence) list of the operators may be required to adjust in order to create another new solution.

Any feasible schedule produced by these scheduling algorithms guarantees that both timing and precedence constraints are met, and the objective of the static scheduler is achieved.

C. FUTURE WORK

This is the first work to implement the static scheduling algorithms for PSDL tasks on multiprocessor systems. Future research is required for identification of possible weaknesses. The continued work is recommended in the following areas:

- **Modifying Proposed Algorithms Using Better Heuristics**

Most heuristic methods suffer from several shortcomings such as the difficulty in assuring the accuracies of solutions [KAN84]. If one algorithm can be proven to be better than the other, which had been proven to be optimal, then this algorithm can also be called optimal. The rule of thumb can be applied to all scientific inventions, including scheduling problems. The better the understanding of the problem, the more opportunities to invent a heuristic solution.

- **Modifying the "Top -Sort" Procedure to Obtain a Better Ordering List**

Since in most hard real-time systems, there exists more than one topological ordering of operators where there are cases in which one ordering may produce a feasible schedule while another will not. It will speed up the existing simulated annealing algorithm if the "top_sort" procedure is good enough to produce a topological ordering of operators such that there is no need to adjust this ordering during the annealing process as frequently as in the present implementation. The improvement of this procedure is recommended by using a similar method of critical path, introduced in [HUS90], with the consideration of giving a weight to each operator.

• **Changing the Assumptions of Scheduling Problem**

Different assumptions can lead to different results. For instance, tasks are assumed to be non-preemptive in this research, but they could be preemptive. There are still many open problems to be considered, such as periodic or non-periodic, any constraints or not, whether the precedence graph is a tree or network, scheduling tasks on centralized or on distributed system, and so forth.

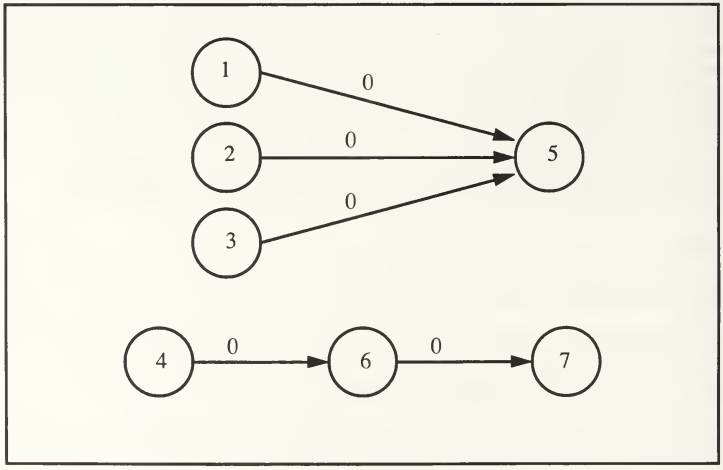
Real-time systems have substantial amounts of knowledge concerning the characteristics of the application and the environment built into the system. A majority of today's systems assume that much of this knowledge is available a priori and, hence, are based on static nature of many of these systems contributes to their high cost and inflexibility. The next generation hard real-time systems must be designed to be dynamic and flexible.

Where as a large proportion of currently implemented hard real-time systems are static in nature, by necessity, next-generation systems will have to adopt solutions that are more dynamic and flexible. This is because we believe that such systems will be large and complex and that they will function in environments that are dynamic while being physically distributed. More important, they will have to be maintainable and extensible due to their evolving nature and projected long lifetimes.

APPENDIX A. EXAMPLES OF TEST DATA

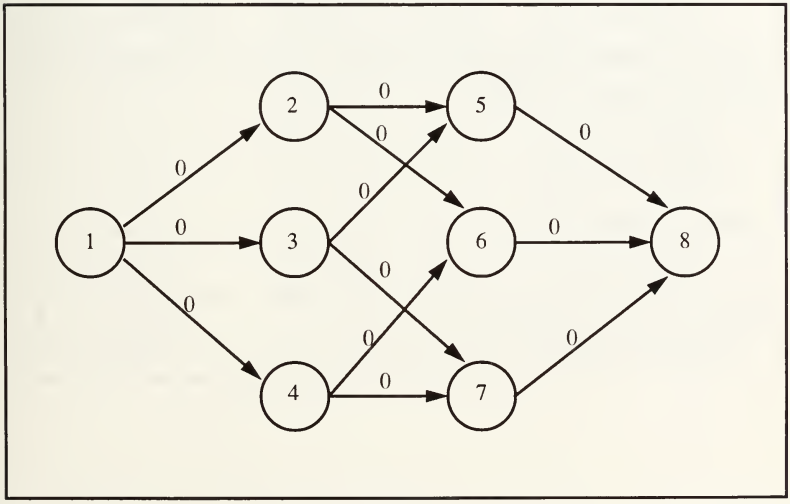
1. Case 1

operator name	No.	MET	Within	P	N
OP_1	1	2000	9000	10000	3
OP_2	2	2000	10000	15000	2
OP_3	3	2000	12000	15000	2
OP_4	4	2000	20000	30000	1
OP_5	5	1000	8000	10000	3
OP_6	6	1000	12000	15000	2
OP_7	7	3000	18000	30000	1



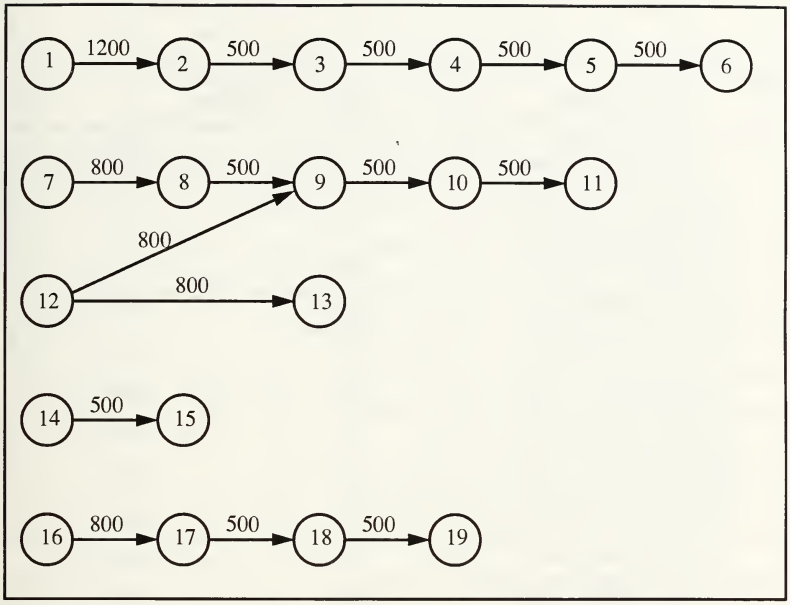
2. Case 2

operator name	No.	MET	Within	P	N
OP_1	1	2000	9000	10000	3
OP_2	2	1000	10000	15000	2
OP_3	3	5000	15000	30000	1
OP_4	4	1000	15000	30000	1
OP_5	5	3000	11000	15000	2
OP_6	6	1000	12000	15000	2
OP_7	7	1000	18000	30000	1
OP_8	8	1000	10000	15000	2



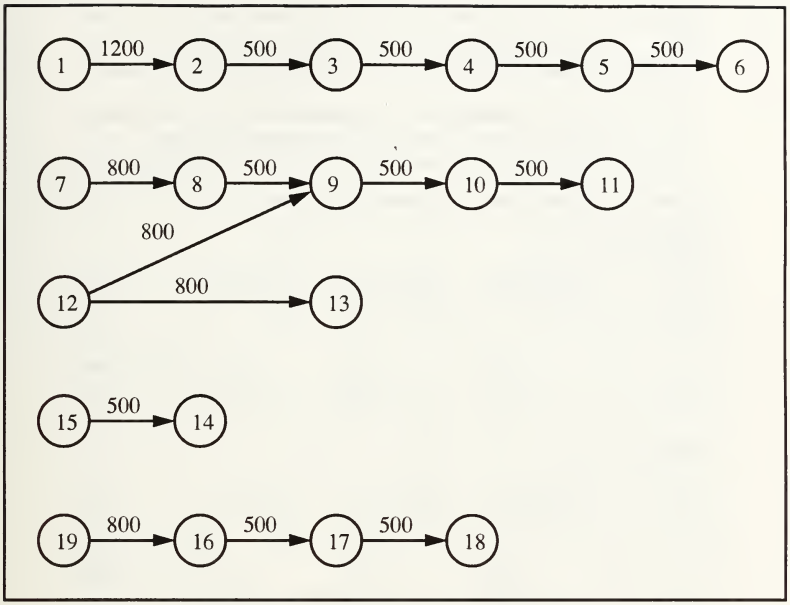
3. Case 3

operator name	No.	MET	Within	P	N
COMMS_LINKS	1	100	-	7000	3
PARSE_INPUT_FILE	2	250	-	7000	3
DECIDE_FOR_ARCHIVING	3	100	-	7000	3
EXTRACT_TRACKS	4	150	-	7000	3
FILTER_COMMS_TRACKS	5	500	-	7000	3
ADD_COMMS_TRACK	6	100	-	7000	3
CREATE_SENSOR_DATA	7	100	-	7000	3
ANALYZE_SENSOR_DATA	8	250	-	7000	3
PREPARE_SENSOR_TRACK	9	250	-	7000	3
FILTER_SENSOR_TRACKS	10	500	-	7000	3
ADD_SENSOR_TRACK	11	500	-	7000	3
CREATE_POSITION_DATA	12	500	-	3000	7
MONITOR_OWNSHIP_POSITION	13	500	-	3000	7
WEAPONS_SYSTEMS	14	100	-	3000	7
WEAPONS_INTERFACE	15	100	-	3000	7
PREPARE_PERIODIC_REPORT	16	500	-	7000	3
MAKE_ROUTING	17	300	-	7000	3
FORWARD_FOR_TRANSMISSION	18	100	-	7000	3
CONVERT_TO_TEXT_FILE	19	100	-	7000	3


1
1200
2
500
3
500
4
500
5
500
6
7
800
8
500
9
500
10
500
11
800
12
800
13
14
500
15
16
800
17
500
18
500
19

4. Case 4

operator name	No.	MET	Within	P	N
COMMS_LINKS	1	1200	-	10000	6
PARSE_INPUT_FILE	2	500	-	10000	6
DECIDE_FOR_ARCHIVING	3	500	-	10000	6
EXTRACT_TRACKS	4	500	-	10000	6
FILTER_COMMS_TRACKS	5	500	-	10000	6
ADD_COMMS_TRACK	6	500	-	10000	6
CREATE_SENSOR_DATA	7	800	-	20000	3
ANALYZE_SENSOR_DATA	8	500	-	20000	3
PREPARE_SENSOR_TRACK	9	500	-	20000	3
FILTER_SENSOR_TRACKS	10	500	-	20000	3
ADD_SENSOR_TRACK	11	500	-	20000	3
CREATE_POSITION_DATA	12	800	-	20000	3
MONITOR_OWNSHIP_POSITION	13	500	-	20000	3
WEAPONS_INTERFACE	14	500	-	5000	12
WEAPONS_SYSTEMS	15	500	-	5000	12
MAKE_ROUTING	16	500	-	15000	4
FORWARD_FOR_TRANSMISSION	17	500	-	15000	4
CONVERT_TO_TEXT_FILE	18	800	-	15000	4
CONVERT_TO_TEXT_FILE	19	800	-	15000	4


1
1200
2
500
3
500
4
500
5
500
6
7
800
8
500
9
500
10
500
11
800
12
800
13
15
500
14
19
800
16
500
17
500
18

APPENDIX B. OUTPUT SCHEDULES FOR TEST DATA

1. Case 1

The HBL := 30000

```
******************************************************
**    the Earliest Starting Time First Scheduling Algorithm    **
******************************************************
```

A feasible solution found; Elapsed time = 0.02 sec.

PROCESSOR # 1

OP	IN	START TIME	STOP TIME	LOWER	UPPER
4	1	0	2000	0	18000
1	2	10000	12000	10000	17000
6	2	17000	18000	17000	28000

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
3	1	0	2000	0	10000
7	1	3000	6000	3000	18000
3	2	15000	17000	15000	25000

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
2	1	0	2000	0	8000
6	1	2000	3000	2000	13000
2	2	15000	17000	15000	23000
5	3	22000	23000	22000	29000

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
1	1	0	2000	0	7000
5	1	2000	3000	2000	9000
5	2	12000	13000	12000	19000
1	3	20000	22000	20000	27000

****
** the Earliest Deadline First Scheduling Algorithm **
****

A feasible schedule found; Elapsed time = 0.03 sec.

PROCESSOR # 1

OP	IN	START TIME	STOP TIME	LOWER	UPPER
5	1	2000	3000	2000	9000
5	2	12000	13000	12000	19000
5	3	22000	23000	22000	29000

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
3	1	0	2000	0	10000
6	1	4000	5000	4000	15000
2	2	15000	17000	15000	23000

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
2	1	0	2000	0	8000
4	1	2000	4000	0	18000
7	1	5000	8000	5000	20000
3	2	15000	17000	15000	25000
6	2	19000	20000	19000	30000

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
1	1	0	2000	0	7000
1	2	10000	12000	10000	17000
1	3	20000	22000	20000	27000

****
** the Simulated Annealing Scheduling Algorithm **
****

A feasible schedule found; Elapsed time = 0.02 sec.

PROCESSOR # 1

OP	IN	START TIME	STOP TIME	LOWER	UPPER
4	1	0	2000	0	18000
6	2	17000	18000	17000	28000

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
3	1	0	2000	0	10000
5	1	2000	3000	2000	9000
2	2	15000	17000	15000	23000

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
2	1	0	2000	0	8000
7	1	3000	6000	3000	18000
1	2	10000	12000	10000	17000
5	2	12000	13000	12000	19000
1	3	20000	22000	20000	27000

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
1	1	0	2000	0	7000
6	1	2000	3000	2000	13000
3	2	15000	17000	15000	25000
5	3	22000	23000	22000	29000

2. Case 2

The HBL := 30000

```
*****************************************************
**    the Earliest Starting Time First Scheduling Algorithm      **
*****************************************************
```

A feasible schedule found; Elapsed time = 0.01 sec. .

PROCESSOR # 1

OP	IN	START TIME	STOP TIME	LOWER	UPPER
4	1	2000	3000	2000	16000
7	1	7000	8000	7000	24000
2	2	17000	18000	17000	26000
8	2	25000	26000	25000	34000

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
3	1	2000	7000	2000	12000
8	1	10000	11000	10000	19000
1	3	20000	22000	20000	27000

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
2	1	2000	3000	2000	11000
5	1	7000	10000	7000	15000
6	2	18000	19000	18000	29000

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
1	1	0	2000	0	7000
6	1	3000	4000	3000	14000
1	2	10000	12000	10000	17000
5	2	22000	25000	22000	30000

****
** the Earliest Deadline First Scheduling Algorithm **
****

A feasible schedule found; Elapsed time = 0.01 sec.

PROCESSOR # 1

OP	IN	START TIME	STOP TIME	LOWER	UPPER
5	1	7000	10000	7000	15000
1	3	20000	22000	20000	27000

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
3	1	2000	7000	2000	12000
8	1	10000	11000	10000	19000
6	2	18000	19000	18000	29000

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
2	1	2000	3000	2000	11000
1	2	10000	12000	10000	17000
5	2	22000	25000	22000	30000

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
1	1	0	2000	0	7000
4	1	2000	3000	2000	16000
6	1	3000	4000	3000	14000
7	1	7000	8000	7000	24000
2	2	17000	18000	17000	26000
8	2	25000	26000	25000	34000

****
** the Simulated Annealing Scheduling Algorithm **
****

A feasible schedule found; Elapsed time = 0.02 sec.

PROCESSOR # 1

OP	IN	START TIME	STOP TIME	LOWER	UPPER
7	1	7000	8000	7000	24000
2	2	17000	18000	17000	26000
1	3	20000	22000	20000	27000

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
3	1	2000	7000	2000	12000
1	2	10000	12000	10000	17000
8	2	25000	26000	25000	34000

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
4	1	2000	3000	2000	16000
6	1	3000	4000	3000	14000
8	1	10000	11000	10000	19000
6	2	18000	19000	18000	29000

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
1	1	0	2000	0	7000
2	1	2000	3000	2000	11000
5	1	7000	10000	7000	15000
5	2	22000	25000	22000	30000

3. Case 3

The HBL := 21000

```
*****************************************************
**    the Earliest Starting Time First Scheduling Algorithm      **
*****************************************************
```

A feasible schedule found; Elapsed time = 0.02 sec.

PROCESSOR # 1

OP	IN	START TIME	STOP TIME	LOWER	UPPER
14	1	0	100	0	2900
8	1	900	1150	900	7650
9	1	1650	1900	1650	8400
4	1	2650	2800	2650	9500
14	2	3000	3100	3000	5900
15	2	3600	3700	3600	6500
6	1	4300	4400	4300	11200
15	3	6600	6700	6600	9500
13	3	7300	7800	7300	9800
9	2	8650	8900	8650	15400
18	2	9200	9300	9200	16100
4	2	9650	9800	9650	16500
13	4	10300	10800	10300	12800
14	5	12000	12100	12000	14900
1	3	14000	14100	14000	20900
14	6	15000	15100	15000	17900
17	3	15400	15700	15400	22100
13	6	16300	16800	16300	18800
11	3	17400	17900	17400	23900
15	7	18600	18700	18600	21500

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
12	1	0	500	0	2500

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
2	1	1300	1550	1300	8050
3	1	2050	2150	2050	8950
19	1	2800	2900	2800	9700
11	1	3400	3900	3400	9900
14	3	6000	6100	6000	8900
7	2	7000	7100	7000	13900
2	2	8300	8550	8300	15050
14	4	9000	9100	9000	11900
10	2	9400	9900	9400	15900
6	2	11300	11400	11300	18200
13	5	13300	13800	13300	15800
8	3	14900	15150	14900	21650
15	6	15600	15700	15600	18500
18	3	16200	16300	16200	23100
19	3	16800	16900	16800	23700
14	7	18000	18100	18000	20900
13	7	193000	19800	19300	21800

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
7	1	0	100	0	6900
15	1	600	700	600	3500
17	1	1400	1700	1400	8100
18	1	2200	2300	2200	9100
12	2	3000	3500	3000	5500
13	2	4300	4800	4300	6800
1	2	7000	7100	7000	13900
8	2	7900	8150	7900	14650
12	4	9000	9500	9000	11500

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
19	2	9800	9900	9800	16700
11	2	10400	10900	10400	16900
15	5	12600	12700	12600	15500
16	3	14100	14600	14100	20600
2	3	15300	15550	15300	22050
3	3	16050	16150	16050	22950
4	3	16650	16800	16650	23500
5	3	17300	17800	17300	23800
6	3	18300	18400	18300	25200

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
1	1	0	100	0	6900
16	1	100	600	0	6500
13	1	1300	1800	1300	3800
10	1	2400	2900	2400	8900
5	1	3300	3800	3300	9800
12	3	6000	6500	6000	8500
16	2	7100	7600	7100	13600
17	2	8400	8700	8400	15100
3	2	9050	9150	9050	15950
15	4	9600	9700	9600	12500
5	2	10300	10800	10300	16800
12	5	12000	12500	12000	14500
7	3	14000	14100	14000	20900
12	6	15000	15500	15000	17500
9	3	15650	15900	15650	22400
10	3	16400	16900	16400	22900
12	7	18000	18500	18000	20500

****
** the Earliest Deadline First Scheduling Algorithm **
****

feasible schedule not found; Elapsed time = 0.02 sec.

PROCESSOR # 1

OP	IN	START TIME	STOP TIME	LOWER	UPPER
13	1	1300	1800	1300	3800
13	2	4300	4800	4300	6800
13	3	7300	7800	7300	9800
13	4	10300	10800	10300	12800
17	2	10800	11100	10800	15700
13	5	13300	13800	13300	15800
18	2	13800	13900	11300	18200
15	6	15600	15700	15600	18500
9	2	15700	15950	13850	20600
14	7	18000	18100	18000	20900
17	3	18100	18400	17500	22700
10	2	18400	18900	16850	23350
18	3	18900	19000	18900	25200
11	2	20100	20600	20100	26600
19	3	21100	21200	21100	28000
11	3	27100	27600	27100	33600

PROCESSORE # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
15	1	600	700	600	3500
16	1	700	1200	0	6500
15	2	3600	3700	3600	6500
14	3	6000	6100	6000	8900
8	1	6100	6350	4100	10850

PROCESSORE # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
2	1	6350	6600	4400	11150
14	4	9000	9100	9000	11900
9	1	9100	9350	6800	13600
3	1	9350	9450	7100	14000
19	1	9450	9550	7100	14000
12	5	12000	12500	12000	14500
4	1	12500	12650	9950	16800
7	2	12650	12750	10200	17100
14	6	15000	15100	15000	17900
11	1	15100	15600	13100	19600
12	7	18000	18500	18000	20500
6	1	18500	18600	16500	23400
4	2	18600	18750	16950	23800
7	3	18750	18850	17200	24100
2	3	20000	20250	20000	25150
3	3	21100	21200	21100	28000
4	3	23950	24100	23950	30800
6	3	30500	30600	30500	37400

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
14	1	0	100	0	2900
12	2	3000	3500	3000	5500
17	1	3500	3800	2000	8700
15	3	6600	6700	6600	9500
15	4	9600	9700	9600	12500
14	5	12000	12100	12000	14900
10	1	12100	12600	9850	16350
1	2	12600	12700	10100	17000

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
8	2	13550	13800	13550	17850
2	2	13900	14150	13900	18150
13	6	16300	16800	16300	18800
13	7	19300	19800	19300	21800
5	2	20150	20650	20150	26650
10	3	23850	24350	23850	30350

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
12	1	0	500	0	2500
14	2	3000	3100	3000	5900
1	1	3100	3200	0	6900
7	1	3200	3300	0	6900
12	3	6000	6500	6000	8500
18	1	6500	6600	4300	11200
12	4	9000	9500	9000	11500
16	2	9500	10000	7700	14200
15	5	12600	12700	12600	15500
12	6	15000	15500	15000	17500
5	1	15500	16000	13150	19650
3	2	16000	16100	14650	21000
19	2	16100	16200	14100	21000
16	3	16200	16700	14700	21200
15	7	18600	18700	18600	21500
1	3	18700	18800	17100	24000
8	3	19650	19900	19650	24850
9	3	20850	21100	20850	27600
6	2	23500	23600	23500	30400
5	3	27150	27650	27150	33650

****
** the Simulated Annealing Scheduling Algorithm **
****

A feasible schedule found; Elapsed time = 0.25 sec.

PROCESSOR # 1

OP	IN	START TIME	STOP TIME	LOWER	UPPER
14	1	0	100	0	2900
17	1	1400	1700	1400	8100
15	1	1700	1800	600	3500
9	1	2550	2800	2550	9300
2	1	2800	3050	1300	8050
4	1	4150	4300	4150	11000
1	2	7000	7100	7000	13900
8	2	7900	8150	7900	14650
14	4	9000	9100	9000	11900
19	2	9800	9900	9800	16700
3	2	10550	10650	10550	17450
5	2	11800	12300	11800	18300
15	5	12600	12700	12600	15500
16	3	14100	14600	14100	20600
12	6	15000	15500	15000	17500
15	6	15600	15700	15600	18500
13	6	16300	16800	16300	18800
11	3	18300	18800	18300	24800
13	7	19300	19800	19300	21800

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
12	1	0	500	0	2500
18	1	2200	2300	2200	9100

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
14	2	3000	3100	3000	5900
15	2	3600	3700	3600	6500
11	1	4300	4800	4300	10800
12	2	4800	5300	3000	5500
12	3	6000	6500	6000	8500
16	2	7100	7600	7100	13600
12	4	9000	9500	9000	11500
15	4	9600	9700	9600	12500
13	4	10300	10800	10300	12800
11	2	11300	11800	11300	17800
12	5	12000	12500	12000	14500
7	3	14000	14100	14000	20900
14	6	15000	15100	15000	17900
18	3	16200	16300	16200	23100
10	3	17300	17800	17300	23800
12	7	18000	18500	18000	20500
15	7	18600	18700	18600	21500
6	3	19800	19900	19800	26700

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
7	1	0	100	0	6900
13	1	1300	1800	1300	3800
8	1	1800	2050	900	7650
10	1	3300	3800	3300	9800
6	1	5800	5900	5800	12700
14	3	6000	6100	6000	8900
15	3	6600	6700	6600	9500
17	2	8400	8700	8400	15100

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
9	2	9550	9800	9550	16300
2	2	9800	10050	8300	15050
4	2	11150	11300	11150	18000
14	5	12000	12100	12000	14900
1	3	14000	14100	14000	20900
8	3	14900	15150	14900	21650
19	3	16800	16900	16800	23700
3	3	17550	17650	17550	24450
14	7	18000	18100	18000	20900
5	3	18800	19300	18800	25300

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
1	1	0	100	0	6900
16	1	100	600	0	6500
19	1	2800	2900	2800	9700
3	1	3550	3650	3550	10450
5	1	4800	5300	4800	11300
13	2	6100	6600	6100	6800
7	2	7000	7100	7000	13900
13	3	7300	7800	7300	9800
18	2	9200	9300	9200	16100
10	2	10300	10800	10300	16800
6	2	12800	12900	12800	19700
13	5	13300	13800	13300	15800
17	3	15400	15700	15400	22100
9	3	16550	16800	16550	23300
2	3	16800	17050	15300	22050
4	3	18150	18300	18150	25000

4. Case 4

The HBL := 60000

```
*****************************************************
**    the Earliest Starting Time First Scheduling Algorithm    **
*****************************************************
```

A feasible solution found; Elapsed time = 0.03 sec.

PROCESSOR # 1

OP	IN	START TIME	STOP TIME	LOWER	UPPER
15	1	0	500	0	4500
19	1	500	1300	0	14200
16	1	2100	2600	2100	16600
3	1	3400	3900	3400	12900
11	1	4600	5100	4600	24100
6	1	6400	6900	6400	15900
2	2	12400	12900	12400	21900
5	2	15400	15900	15400	24900
16	2	17100	17600	17100	31600
12	2	20000	20800	20000	39200
13	2	21600	22100	21600	41100
10	2	23600	24100	23600	43100
5	3	25400	25900	25400	34900
15	7	30000	30500	30000	34500
16	3	32100	32600	32100	46600
18	3	34100	34900	34100	48300
6	4	36400	36900	36400	45900
15	9	40000	40500	40000	44500
14	9	41000	41500	41000	45500
9	3	42600	43100	42600	62100
11	3	44600	45100	44600	64100
14	10	46000	46500	46000	50500
18	4	49100	49900	49100	63300

PROCESSOR # 1

OP	IN	START TIME	STOP TIME	LOWER	UPPER
2	6	52400	52900	52400	61900
5	6	55400	55900	55400	64900

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
12	1	0	800	0	19200
8	1	1600	2100	1600	21100
17	1	3100	3600	3100	17600
4	1	4400	4900	4400	13900
14	2	6000	6500	6000	10500
14	3	11000	11500	11000	15500
15	4	15000	15500	15000	19500
6	2	16400	16900	16400	25900
7	2	20000	20800	20000	39200
8	2	21600	22100	21600	41100
3	3	23400	23900	23400	32900
15	6	25000	25500	25000	29500
1	4	30000	31200	30000	38800
2	4	32400	32900	32400	41900
4	4	34400	34900	34400	43900
14	8	36000	36500	36000	40500
1	5	40000	41200	40000	48800
2	5	42400	42900	42400	51900
4	5	44400	44900	44400	53900
19	4	45500	46300	45500	59700
17	4	48100	48600	48100	62600
14	11	51000	51500	51000	55500
15	12	55000	55500	55000	59500

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
7	1	0	800	0	19200
14	1	1000	1500	1000	5500
2	1	2400	2900	2400	11900
10	1	3600	4100	3600	23100
15	2	5000	5500	5000	9500
1	2	10000	11200	10000	18800
4	2	14400	14900	14400	23900
14	4	16000	16500	16000	20500
18	2	19100	19900	19100	33300
15	5	20000	20500	20000	24500
14	5	21000	21500	21000	25500
9	2	22600	23100	22600	42100
11	2	24600	25100	24600	44100
6	3	26400	26900	26400	35900
14	7	31000	31500	31000	35500
3	4	33400	33900	33400	42900
5	4	35400	35900	35400	44900
12	3	40000	40800	40000	59200
13	3	41600	42100	41600	61100
10	3	43600	44100	43600	63100
5	5	45400	45900	45400	54900
16	4	47100	47600	47100	61600
15	11	50000	50500	50000	54500
3	6	53400	53900	53400	62900
14	12	56000	56500	56000	60500

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
1	1	0	1200	0	8800

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
13	1	1600	2100	1600	21100
9	1	2600	3100	2600	22100
18	1	4100	4900	4100	18300
5	1	5400	5900	5400	14900
15	3	10000	10500	10000	14500
3	2	13400	13900	13400	22900
19	2	15500	16300	15500	29700
17	2	18100	18600	18100	32600
1	3	20000	21200	20000	28800
2	3	22400	22900	22400	31900
4	3	24400	24900	24400	33900
14	6	26000	26500	26000	30500
19	3	30500	31300	30500	44700
17	3	33100	33600	33100	47600
15	8	35000	35500	35000	39500
7	3	40000	40800	40000	59200
8	3	41600	42100	41600	61100
3	5	43400	43900	43400	52900
15	10	45000	45500	45000	49500
6	5	46400	46900	46400	55900
1	6	50000	51200	50000	58800
4	6	54400	54900	54400	63900
6	6	56400	56900	56400	65900

```
*****************************************************
**        the Earliest Deadline First Scheduling Algorithm        **
*****************************************************
```

feasible schedule not found; Elapsed time = 0.03 sec.

PRECESSOR # 1

OP	IN	START TIME	STOP TIME	LOWER	UPPER
15	2	5000	5500	5000	9500
14	3	11000	11500	11000	15500
3	2	13400	13900	13400	22900
15	5	20000	20500	20000	24500
14	6	26000	26500	26000	30500
9	1	26500	27000	17900	37400
15	8	35000	35500	35000	39500
14	9	41000	41500	41000	45500
3	5	43400	43900	43400	52900
15	11	50000	50500	50000	54500
14	12	56000	56500	56000	60500
8	3	57100	57600	57100	68000
11	2	58700	59200	58700	78200

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
1	1	0	1200	0	8800
2	1	2400	2900	2400	11900
4	1	4400	4900	4400	13900
5	1	5400	5900	5400	14900
6	1	6400	6900	6400	15900
7	1	6900	7700	0	19200
12	1	7700	8500	0	19200
15	4	15000	15500	15000	19500

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
14	5	21000	21500	21000	25500
2	3	22400	22900	22400	31900
19	2	22900	23700	18900	33100
4	3	24400	24900	24400	33900
5	3	25400	25900	25400	34900
14	7	31000	31500	31000	35500
2	4	32400	32900	32400	41900
4	4	34400	34900	34400	43900
5	4	35400	35900	35400	44900
6	4	36400	36900	36400	45900
18	2	36900	37700	32500	46700
10	1	37700	38200	27500	47000
19	3	38200	39000	33900	48100
1	5	40000	41200	40000	48800
2	5	42400	42900	42400	51900
4	5	44400	44900	44400	53900
5	5	45400	45900	45400	54900
6	5	46400	46900	46400	55900
9	2	46900	47400	39100	57400
1	6	50000	51200	50000	58800
18	3	51200	52000	47500	61700
3	6	53400	53900	53400	62900
5	6	55400	55900	55400	64900
12	3	55900	56700	47700	66900
13	3	57500	58000	57500	68800
10	3	67500	68000	67500	87000

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
14	1	1000	1500	1000	5500
3	1	3400	3900	3400	12900
19	1	3900	4700	0	14200
15	3	10000	10500	10000	14500
16	1	10500	11000	5500	20000
14	4	16000	16500	16000	20500
17	1	16500	17000	11500	26000
1	3	20000	21200	20000	28800
18	1	21200	22000	17500	31700
3	3	23400	23900	23400	32900
15	7	30000	30500	30000	34500
14	8	36000	36500	36000	40500
7	2	36500	37300	26900	46100
12	2	37300	38100	27700	46900
8	2	38100	38600	38100	48000
13	2	38900	39400	38900	48800
15	10	45000	45500	45000	49500
14	11	51000	51500	51000	55500
2	6	52400	52900	52400	61900
19	4	52900	53700	48900	63100
4	6	54400	54900	54400	63900
16	4	54900	55400	54500	65000
6	6	56400	56900	56400	65900
18	4	62500	63300	62500	76700

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
15	1	0	500	0	4500
14	2	6000	6500	6000	10500

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
1	2	10000	11200	10000	18800
2	2	12400	12900	12400	21900
4	2	14400	14900	14400	23900
5	2	15400	15900	15400	24900
6	2	16400	16900	16400	25900
8	1	16900	17400	8500	28000
13	1	17400	17900	9300	28800
15	6	25000	25500	25000	29500
16	2	25500	26000	24500	35000
6	3	26400	26900	26400	35900
1	4	30000	31200	30000	38800
17	2	31200	31700	26500	41000
3	4	33400	33900	33400	42900
15	9	40000	40500	40000	44500
16	3	40500	41000	39800	50000
14	10	46000	46500	46000	50500
17	3	46500	47000	41500	56000
11	1	47000	47500	38700	58200
15	12	55000	55500	55000	59500
7	3	55500	56300	46900	66100
10	2	56300	56800	47500	67000
17	4	56800	57300	56500	71000
9	3	57900	58400	57900	77400
11	3	78700	79200	78700	98200

****
** the Simulated Annealing Scheduling Algorithm **
****

A feasible schedule found; Elapsed time = 0.12 sec.

PROCESSOR # 1

OP	IN	START TIME	STOP TIME	LOWER	UPPER
15	1	0	500	0	4500
19	1	500	1300	0	14200
14	1	1300	1800	1000	5500
13	1	1800	2300	1600	21100
8	1	2300	2800	1600	21100
10	1	4300	4800	4300	23800
4	1	5800	6300	5800	15300
16	2	17100	17600	17100	31600
12	2	20000	20800	20000	39200
14	5	21000	21500	21000	25500
10	2	24300	24800	24300	43800
15	6	25000	25500	25000	29500
14	6	26000	26500	26000	30500
16	3	32100	32600	32100	46600
15	8	35000	35500	35000	39500
14	8	36000	36500	36000	40500
12	3	40000	40800	40000	59200
14	9	41000	41500	41000	45500
10	3	44300	44800	44300	63800
3	5	44800	45300	44800	54300
14	10	46000	46500	46000	50500
17	4	48100	48600	48100	62600

PROCESSOR # 2

OP	IN	START TIME	STOP TIME	LOWER	UPPER
12	1	0	800	0	19200
17	1	3100	3600	3100	17600
11	1	5300	5800	5300	24800
14	2	6000	6500	6000	10500
19	2	15500	16300	15500	29700
5	2	16800	17300	16800	26300
6	2	17800	18300	17800	27300
1	3	20000	21200	20000	28800
13	2	21600	22100	21600	41100
2	3	22400	22900	22400	31900
3	3	24800	25300	24800	34300
6	3	27800	28300	27800	37300
15	7	30000	30500	30000	34500
14	7	31000	31500	31000	35500
17	3	33100	33600	33100	47600
7	3	40000	40800	40000	59200
9	3	43300	43800	43300	62800
15	10	45000	45500	45000	49500
16	4	47100	47600	47100	61600
18	4	49100	49900	49100	63300
3	6	54800	55300	54800	64300
4	6	55800	56300	55800	65300
5	6	56800	57300	56800	66300

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
7	1	0	800	0	19200
16	1	2100	2600	2100	16600
9	1	3300	3800	3300	22800

PROCESSOR # 3

OP	IN	START TIME	STOP TIME	LOWER	UPPER
2	1	3800	4300	2400	11900
3	1	4800	5300	4800	14300
6	1	7800	8300	7800	17300
1	2	10000	11200	10000	18800
2	2	12400	12900	12400	21900
3	2	14800	15300	14800	24300
4	2	15800	16300	15800	15300
18	2	19100	19900	19100	33300
15	5	20000	20500	20000	24500
8	2	21600	22100	21600	41100
11	2	25300	25800	25300	44800
4	3	25800	26300	25800	35300
5	3	26800	27300	26800	36300
1	4	30000	31200	30000	38800
18	3	34100	34900	34100	48300
4	4	35800	36300	35800	45300
1	5	40000	41200	40000	48800
8	3	41600	42100	41600	61100
2	5	42400	42900	42400	51900
19	4	45500	46300	45500	59700
1	6	50000	51200	50000	58800
2	6	52400	52900	52400	61900
14	12	56000	56500	56000	60500
6	6	57800	58300	57800	67300

PROCESSOR # 4

OP	IN	START TIME	STOP TIME	LOWER	UPPER
1	1	0	1200	0	8800
18	1	4100	4900	4100	18300
15	2	5000	5500	5000	9500
5	1	6800	7300	6800	16300
15	3	10000	10500	10000	14500
14	3	11000	11500	11000	15500
15	4	15000	15500	15000	19500
14	4	16000	16500	16000	20500
17	2	18100	18600	18100	32600
7	2	20000	20800	20000	39200
9	2	23300	23800	23300	42800
19	3	30500	31300	30500	44700
2	4	32400	32900	32400	41900
3	4	34800	35300	34800	44300
5	4	36800	37300	36800	46300
6	4	37800	38300	37800	47300
15	9	40000	40500	40000	44500
13	3	41600	42100	41600	61100
11	3	45300	45800	45300	64800
4	5	45800	46300	45800	55300
5	5	46800	47300	46800	56300
6	5	47800	48300	47800	57300
15	11	50000	50500	50000	54500
14	11	51000	51500	51000	55500
15	12	55000	55500	55000	59500

APPENDIX C. ADA CODES OF THE MODIFIED PACKAGES

1. DATA

```
with VSTRINGS;
with SEQUENCES;
with TEXT_IO;
        -- This package contains all of the global declarations and definitions
        --      of data structures that are necessary for the Static Scheduler
package DATA is
  package VARSTRING is new VSTRINGS(80);
  use VARSTRING;
  subtype OPERATOR_ID is VSTRING;
  subtype VALUE is NATURAL;
  subtype MET is VALUE;
  subtype MRT is VALUE;
  subtype MCP is VALUE;
  subtype PERIOD is VALUE;
  subtype WITHIN is VALUE;
  subtype STARTS is VALUE;
  subtype STOPS is VALUE;
  subtype LOWERS is VALUE;
  subtype UPPERS is VALUE;
  Exception_Operator: OPERATOR_ID;
  TEST_VERIFIED   : BOOLEAN   :=  true;
  NOP             : NATURAL   :=  4; -- number of available processors (6/25/92)
  type PROCESSOR_ARRAY is array (1..NOP) of VALUE; --(8/12/92)
  type OPERATOR is
    record
      THE_OPERATOR_ID   :   OPERATOR_ID;
      THE_MET           :   MET    :=  0;
      THE_MRT           :   MRT    :=  0;
      THE_MCP           :   MCP    :=  0;
      THE_PERIOD        :   PERIOD :=  0;
      THE_WITHIN        :   WITHIN :=  0;
    end record;

  package V_LISTS is new SEQUENCES(OPERATOR); use V_LISTS;
```

```
type SCHEDULE_INPUTS is
  record
    THE_OPERATOR      :   INTEGER;
    THE_START         :   STARTS    :=  0;
    THE_STOP          :   STOPS     :=  0;
    THE_LOWER         :   LOWERS    :=  0;
    THE_UPPER         :   UPPERS    :=  0;
    THE_INSTANCE      :   INTEGER   :=  1;
  end record;
package SCHEDULE_INPUTS_LIST is new SEQUENCES(SCHEDULE_INPUTS);

type SCHEDULE_ARRAY is array (1..NOP) of SCHEDULE_INPUTS_LIST.LIST;-- (7/10/92)

package NODE_LIST is new SEQUENCES(INTEGER);

NON_CRITS    :   TEXT_IO.FILE_TYPE;
AG_OUTFILE   :   TEXT_IO.FILE_TYPE;
INPUT        :   TEXT_IO.FILE_MODE    :=  TEXT_IO.IN_FILE;
OUTPUT       :   TEXT_IO.FILE_MODE    :=  TEXT_IO.OUT_FILE;
Current_Value :  VALUE;
New_Word     :   VARSTRING.VSTRING;
Cur_Opt      :   OPERATOR;
OP_COUNT     :   INTEGER;
OP_LIST      :   V_LISTS.LIST;
-- the following global variables are new for multiprocessor scheduling (7/30/92)
OP_NUM           :   INTEGER;
PARENT_COUNT     :   INTEGER;
PARENT_NUM       :   INTEGER;
CHILD_COUNT      :   INTEGER;
PROCESSOR_NUM    :   INTEGER;
PARENT_OP        :   OPERATOR;
PARENT_LIST      :   NODE_LIST.LIST;
CHILD_LIST       :   NODE_LIST.LIST;
LIST_HEAD        :   NODE_LIST.LIST;
TEMP             :   OPERATOR;
NEW_INPUT        :   SCHEDULE_INPUTS;
ADDL_NODE        :   SCHEDULE_INPUTS;
FIRST            :   SCHEDULE_INPUTS;
BEST             :   SCHEDULE_INPUTS;

end DATA;
```

2. NEW_DATA_STRUCTURES

```
with TEXT_IO;
with DATA; use DATA;

-- This package contains the specifications for a graph data structure that can represent an
-- acyclic graph. Functions and procedures exist to access the information that is stored in the graph
-- as well as to find out the relationships between vertices in the graph.

generic

package NEW_DATA_STRUCTURES is

  type GRAPH (SIZE: INTEGER) is limited private;
  type GRAPH_LINK is access GRAPH;
  THE_GRAPH        : GRAPH_LINK;

  procedure PRODUCE_OP_ARRAY ( INFO_LIST   : in out V_LISTS.LIST;
                               COUNT       : in INTEGER);
    -- Transfer operator info from linked list to array

  function OP_POSITION ( OP_NAME    : VARSTRING.VSTRING;
                         COUNT      : INTEGER) return INTEGER;
    -- Given an operator name return the operator's position in the array

  procedure PRODUCE_OP_MATRIX (COUNT     : in INTEGER);
    -- Create a matrix to represent the acyclic graph of operator relationship

  function OP_RETURN (OP_POSITION: INTEGER) return OPERATOR;
    -- Given an operator's position in the array, return the operator

  function IS_PARENT (OP_1, OP_2: INTEGER) return BOOLEAN;
    -- Return true if OP_1 is a parent of OP_2 or if OP_1 is OP_2

  function IS_CHILD (OP_1, OP_2: INTEGER) return BOOLEAN;
    -- Return true if OP_1 is a child of OP_2 or if OP_1 is OP_2

  procedure RETURN_PARENT_LIST ( PARENT_LIST : in out NODE_LIST.LIST;
                                 OP          : in INTEGER;
                                 COUNT       : in out INTEGER);
    -- Return a list of all the parents of an operator
```

```
  procedure RETURN_CHILD_LIST ( CHILD_LIST      : in out NODE_LIST.LIST;
                                OP              : in INTEGER;
                                COUNT           : in out INTEGER);
    -- Return a list of all the children of an operator

  procedure FREE_GRAPH (A_GRAPH     : in out GRAPH_LINK);
    -- Free the memory space used by the graph

  function LATENCY(OP1,OP2: INTEGER) return INTEGER; -- (7/10/92)

  function PIPELINE(OP: INTEGER) return BOOLEAN; -- (7/10/92)

  private

  type INFO_ARRAY is array (INTEGER range <>) of OPERATOR;

  type MATRIX_OP_INFO is
    record
      PARENT            : INTEGER   :=  -1;
      CHILD             : INTEGER   :=  -1;
      DELAY_PIPELINE    : INTEGER   :=  0; --(7/10/92)
    end record;

  type MATRIX is array (INTEGER range <>,INTEGER range <>) of MATRIX_OP_INFO;

  type GRAPH (SIZE: INTEGER) is
    record
      OP_ARRAY   : INFO_ARRAY(0..SIZE);
      OP_MATRIX  : MATRIX(0..SIZE, 0..SIZE);
    end record;

end NEW_DATA_STRUCTURES;
```

```
with UNCHECKED_DEALLOCATION;
with TEXT_IO; use TEXT_IO;

package body NEW_DATA_STRUCTURES is

  pragma LINK_WITH ("heaplib.sparc.ar");

  procedure FREE is new UNCHECKED_DEALLOCATION(GRAPH, GRAPH_LINK);

  package int_io is new TEXT_IO.INTEGER_IO(INTEGER); use int_io;

  procedure PRODUCE_OP_ARRAY ( INFO_LIST   : in out V_LISTS.LIST;
                               COUNT       : in INTEGER) is

    HEAD       :V_LISTS.LIST := INFO_LIST;

    function MAKE_START_NODE return OPERATOR is

      START_OP    : OPERATOR;

    begin
      START_OP.THE_OPERATOR_ID    :=  VARSTRING.VSTR("DUMMY START NODE");
      START_OP.THE_MET            :=  0;
      START_OP.THE_MRT            :=  0;
      START_OP.THE_MCP            :=  0;
      START_OP.THE_WITHIN         :=  0;
      return START_OP;
    end MAKE_START_NODE;

  begin
      for INDEX in reverse 1..COUNT loop
        THE_GRAPH.OP_ARRAY(INDEX) := V_LISTS.VALUE(INFO_LIST);
        V_LISTS.NEXT(INFO_LIST);
      end loop;
      THE_GRAPH.OP_ARRAY(0) := MAKE_START_NODE;
      V_LISTS.FREE_LIST(HEAD); --* THIS LIST IS NO LONGER NEEDED.

  end PRODUCE_OP_ARRAY;
```

```
function OP_POSITION ( OP_NAME : VARSTRING.VSTRING;
                       COUNT   : INTEGER                ) return INTEGER is
begin
  for INDEX in 1..COUNT loop
    if VARSTRING.EQUAL(OP_NAME, THE_GRAPH.OP_ARRAY(INDEX).
          THE_OPERATOR_ID) then return INDEX;
    end if;
  end loop;
  return -1; -- Operator is external since it is not in the array.
end OP_POSITION;

procedure PRODUCE_OP_MATRIX (COUNT:in INTEGER) is

  COLUMN,
  ROW,
  PARENT_OP,
  CHILD_OP     : INTEGER;
  LINK         : constant VARSTRING.VSTRING := VARSTRING.VSTR("LINK");

  procedure INITIALIZE ( COUNT     : in INTEGER;
                         OP_MATRIX : in out MATRIX) is
  begin
    for ROW in 0..COUNT loop
      THE_GRAPH.OP_MATRIX(ROW,ROW).PARENT := ROW;
      THE_GRAPH.OP_MATRIX(ROW,ROW).CHILD := ROW;
    end loop;
  end INITIALIZE; --(6/5/92, S)

  procedure INITIALIZE_START_NODE ( COUNT     : in INTEGER;
                                    OP_MATRIX : in out MATRIX) is
  begin
    for INDEX in 0..COUNT loop
      if THE_GRAPH.OP_MATRIX(INDEX, INDEX).PARENT = INDEX then
        THE_GRAPH.OP_MATRIX(INDEX,INDEX).PARENT := 0;
        THE_GRAPH.OP_MATRIX(0,INDEX).CHILD := THE_GRAPH.OP_MATRIX(0,0).CHILD;
        THE_GRAPH.OP_MATRIX(0,0).CHILD := INDEX;
      THE_GRAPH.OP_MATRIX(0,INDEX).PARENT := INDEX;
      end if;
    end loop;
  end INITIALIZE_START_NODE;
```

```
begin -- PRODUCE_OP_MATRIX
  TEXT_IO.OPEN (AG_OUTFILE,INPUT,"atomic.info");
  INITIALIZE(COUNT, THE_GRAPH.OP_MATRIX);
  VARSTRING.GET_LINE (AG_OUTFILE, New_Word);
  while not TEXT_IO.END_OF_FILE(AG_OUTFILE) loop
    if VARSTRING.EQUAL (New_Word,LINK) then          -- keyword "LINK"
      TEXT_IO.SKIP_LINE(AG_OUTFILE);                 -- skip LINK name
      VARSTRING.GET_LINE(AG_OUTFILE, New_Word);
      PARENT_OP := OP_POSITION (NEW_WORD, DATA.OP_COUNT);
      int_io.GET(AG_OUTFILE,Current_Value); --(7/28/92)
      TEXT_IO.SKIP_LINE(AG_OUTFILE);
      VARSTRING.GET_LINE (AG_OUTFILE, New_Word);
      CHILD_OP := OP_POSITION(New_Word, DATA.OP_COUNT);
      if (PARENT_OP /= -1 and CHILD_OP /= -1) then
        if THE_GRAPH.OP_MATRIX(PARENT_OP,CHILD_OP).DELAY_PIPELINE
            < Current_Value then
          THE_GRAPH.OP_MATRIX(PARENT_OP,CHILD_OP).DELAY_PIPELINE
            = Current_Value;
        end if;
        if (THE_GRAPH.OP_MATRIX(PARENT_OP,CHILD_OP).CHILD = -1) then
          THE_GRAPH.OP_MATRIX(PARENT_OP,CHILD_OP).CHILD
            := THE_GRAPH.OP_MATRIX(PARENT_OP,PARENT_OP).CHILD;
          THE_GRAPH.OP_MATRIX(PARENT_OP,PARENT_OP).CHILD := CHILD_OP;
          THE_GRAPH.OP_MATRIX(PARENT_OP,CHILD_OP).PARENT
            := THE_GRAPH.OP_MATRIX(CHILD_OP,CHILD_OP).PARENT;
          THE_GRAPH.OP_MATRIX(CHILD_OP,CHILD_OP).PARENT := PARENT_OP;
        end if;
      end if;
    end if;
    VARSTRING.GET_LINE(AG_OUTFILE, New_Word);
  end loop;
  TEXT_IO.CLOSE (AG_OUTFILE);
  INITIALIZE_START_NODE(COUNT, THE_GRAPH.OP_MATRIX);
end PRODUCE_OP_MATRIX;

function OP_RETURN (OP_POSITION: INTEGER) return OPERATOR is
  OP: OPERATOR;
begin
  OP := THE_GRAPH.OP_ARRAY(OP_POSITION);
  return OP;
end OP_RETURN;
```

```
function IS_PARENT (OP_1, OP_2: INTEGER) return BOOLEAN is
          -- Return true if OP_1 is a parent of OP_2 or if OP_1 is OP_2

  PARENT: BOOLEAN := false;

begin
  if OP_1 = OP_2 then
    PARENT := true;
  elsif THE_GRAPH.OP_MATRIX(OP_1, OP_2).PARENT /= -1 then
      PARENT := true;
  end if;
  return PARENT;
end IS_PARENT;

function IS_CHILD (OP_1, OP_2: INTEGER) return BOOLEAN is
          -- Return true if OP_1 is a child of OP_2 or if OP_1 is OP_2
  CHILD: BOOLEAN := false;

begin
  if OP_1 = OP_2 then
    CHILD := true;
  elsif THE_GRAPH.OP_MATRIX(OP_2, OP_1).CHILD /= -1 then
    CHILD := true;
  end if;
  return CHILD;
end IS_CHILD;

procedure RETURN_PARENT_LIST ( PARENT_LIST  : in out NODE_LIST.LIST;
                               OP           : in INTEGER;
                               COUNT        : in out INTEGER) is
    ROW: INTEGER := OP;

begin
  COUNT := 0;
  while THE_GRAPH.OP_MATRIX(ROW, OP).PARENT /= OP loop
    NODE_LIST.ADD(THE_GRAPH.OP_MATRIX(ROW, OP).PARENT, PARENT_LIST);
    COUNT := COUNT + 1;
    ROW := THE_GRAPH.OP_MATRIX(ROW, OP).PARENT;
  end loop;
end RETURN_PARENT_LIST;
```

```
  procedure RETURN_CHILD_LIST ( CHILD_LIST : in out NODE_LIST.LIST;
                                OP         : in INTEGER;
                                COUNT      : in out INTEGER) is

      COLUMN: INTEGER := OP;

  begin
    COUNT := 0;
    while THE_GRAPH.OP_MATRIX(OP, COLUMN).CHILD /= OP loop
      NODE_LIST.ADD(THE_GRAPH.OP_MATRIX(OP, COLUMN).CHILD, CHILD_LIST);
      COUNT := COUNT + 1;
      COLUMN := THE_GRAPH.OP_MATRIX(OP, COLUMN).CHILD;
    end loop;
  end RETURN_CHILD_LIST;

  procedure FREE_GRAPH (A_GRAPH     : in out GRAPH_LINK) is

  begin
    FREE(A_GRAPH);
  end FREE_GRAPH;

  function LATENCY (OP1, OP2: INTEGER) return INTEGER is --(7/10/92)

  begin
    return THE_GRAPH.OP_MATRIX(OP1,OP2).DELAY_PIPELINE;
  end LATENCY;

  function PIPELINE(OP: INTEGER) return BOOLEAN is --(7/10/92)

  begin
    if THE_GRAPH.OP_MATRIX(OP,OP).DELAY_PIPELINE = 1 then
      return true;
    else
      return false;
    end if;
  end PIPELINE;


end NEW_DATA_STRUCTURES;
```

3. DIAGNOSTICS

```
with DATA; use DATA;

package DIAGNOSTICS is

  procedure OUTPUT_SCHEDULE (AGENDA: in SCHEDULE_ARRAY); --(7/21/92)

  procedure OUTPUT_OP_ID (OP_EL: in V_LISTS.LIST);

  procedure OUTPUT_HARMONIC_BLOCK_LENGTH (H_B_LENGTH: in INTEGER);

  procedure OUTPUT_PRECEDENCE_LIST (PREC_LIST: in NODE_LIST.LIST);

end DIAGNOSTICS;

with DATA;          use DATA;
with TEXT_IO;       use TEXT_IO; --(7/21/92)
with FRONT_END;     use FRONT_END; --(7/21/92)

package body DIAGNOSTICS is

  package int_io is new integer_io(integer); use int_io;

  procedure OUTPUT_SCHEDULE (AGENDA: in SCHEDULE_ARRAY) is --(7/21/92)

    AGENDA_1: SCHEDULE_ARRAY := AGENDA; --(7/21/92)

  begin
    for PROCESSOR_NUM in 1..NOP loop
      SET_COL(1);
      PUT("THE PROCESSOR NO. : ");
      int_io.PUT(PROCESSOR_NUM, WIDTH=>2);
      NEW_LINE;
      SET_COL(3);
      PUT("OP#");
      SET_COL(8);
      PUT("INSTANCE # ");
      SET_COL(20);
      PUT("START TIME");
      SET_COL(32);
```

```
    PUT("STOP TIME");
    SET_COL(46);
    PUT("LOWER");
    SET_COL(56);
    PUT("UPPER ");
    NEW_LINE;

    while SCHEDULE_INPUTS_LIST.NON_EMPTY(AGENDA_1(PROCESSOR_NUM)) loop
      SET_COL(3);
      int_io.PUT(SCHEDULE_INPUTS_LIST.VALUE(AGENDA_1(PROCESSOR_NUM)).
            THE_OPERATOR, WIDTH => 3);
      SET_COL(6);
      PUT("-");
      SET_COL(7);
      int_io.PUT(SCHEDULE_INPUTS_LIST.VALUE(AGENDA_1(PROCESSOR_NUM)).
            THE_INSTANCE, width => 10);
      SET_COL(20);
      int_io.PUT(SCHEDULE_INPUTS_LIST.VALUE(AGENDA_1(PROCESSOR_NUM)).
            THE_START, width => 10);
      SET_COL(31);
      int_io.PUT(SCHEDULE_INPUTS_LIST.VALUE(AGENDA_1(PROCESSOR_NUM)).
            THE_STOP, width => 10);
      SET_COL(41);
      int_io.PUT(SCHEDULE_INPUTS_LIST.VALUE(AGENDA_1(PROCESSOR_NUM)).
            THE_LOWER, width => 10);
      SET_COL(51);
      int_io.PUT(SCHEDULE_INPUTS_LIST.VALUE(AGENDA_1(PROCESSOR_NUM)).
            THE_UPPER, width => 10);
      NEW_LINE;
      SCHEDULE_INPUTS_LIST.NEXT(AGENDA_1(PROCESSOR_NUM));
    end loop;
  end loop;
end OUTPUT_SCHEDULE;

procedure OUTPUT_HARMONIC_BLOCK_LENGTH (H_B_LENGTH: in INTEGER) is

begin
  PUT("The Harmonic Block Length is:");
  int_io.PUT(H_B_LENGTH);
  NEW_LINE;
end OUTPUT_HARMONIC_BLOCK_LENGTH;
```

```
  procedure OUTPUT_OP_ID (OP_EL: in V_LISTS.LIST) is

    TRAVERSE: V_LISTS.LIST := OP_EL;

  begin
    VARSTRING.PUT(V_LISTS.VALUE(TRAVERSE).THE_OPERATOR_ID);
  end OUTPUT_OP_ID;

  procedure OUTPUT_PRECEDENCE_LIST (PREC_LIST: in NODE_LIST.LIST) is

    TRAVERSE: NODE_LIST.LIST := PREC_LIST;

  begin
    while NODE_LIST.NON_EMPTY(TRAVERSE) loop
      VARSTRING.PUT(NEW_GRAPH.OP_RETUR(NODE_LIST.VALUE(TRAVERSE)).
            THE_OPERATOR_ID);
      NEW_LINE;
      NODE_LIST.NEXT(TRAVERSE);
    end loop;
  end OUTPUT_PRECEDENCE_LIST;


end DIAGNOSTICS;
```

APPENDIX D. ADA CODES OF THE NEW PACKAGES

1. UTILITY_PKG

```
with DATA; use DATA;

package UTILITY_PKG is

  M     : CONSTANT := 2**13;
  subtype NUMBER is FLOAT range 0.0..1.0;
  subtype SEED is INTEGER range 1..M-1;

  procedure RANDOM_INITIALIZE(START_VALUE: in SEED); -- initialize random number generator.

  function RANDOM_NEXT return NUMBER; -- gives a random number between 0 and 1.

  procedure DETERMINE_THE_UPPER(TEMP          : in OPERATOR;
                                NEW_NODE      : in out SCHEDULE_INPUTS);

  procedure DETERMINE_START_STOP( TEMP        : in OPERATOR;
                                  PROC_NUM    : in out INTEGER;
                                  NEW_NODE    : in out SCHEDULE_INPUTS;
                                  PROC_STOP   : in out PROCESSOR_ARRAY);

  procedure CREATE_ADDL_NODE(NEW_NODE     : in SCHEDULE_INPUTS;
                             TEMP         : in OPERATOR;
                             ADDL_NODE    : in out SCHEDULE_INPUTS);

  procedure TEST_SCHEDULE( HBL       : in INTEGER
                           AGENDA    : in SCHEDULE_ARRAY;
                           COST      : in out INTEGER);

  function ANNEAL_FUNCTION( COST_1           : INTEGER;
                            COST_2           : INTEGER;
                            CURRENT_TEMPER   : FLOAT     ) return FLOAT;

  procedure ADJUST_PRECEDENCE( COUNT  : in INTEGER;
                               P_LIST : in out NODE_LIST.LIST);

end UTILITY_PKG;
```

```
with DATA; use DATA;
with FRONT_END; use FRONT_END;
with PRIORITY_QUEUES;
with MATH; use MATH;
  with TEXT_IO; use TEXT_IO;
  with DIAGNOSTICS;

package body UTILITY_PKG is

  U : NATURAL;
  K : CONSTANT := 5**5;

  package  int_io is new TEXT_IO.INTEGER_IO(INTEGER);

  procedure RANDOM_INITIALIZE(START_VALUE : in SEED) is

  begin
    U := START_VALUE;
  end RANDOM_INITIALIZE;

  function RANDOM_NEXT return NUMBER is

  begin
    U := U * K mod M;
    return FLOAT(U)/FLOAT(M);
  end RANDOM_NEXT;

  procedure DETERMINE_THE_UPPER(TEMP          : in OPERATOR;
                                NEW_NODE      : in out SCHEDULE_INPUTS) is

  begin

    if   TEMP.THE_WITHIN /= 0 then
         NEW_NODE.THE_UPPER :=
              NEW_NODE.THE_LOWER + TEMP.THE_WITHIN - TEMP.THE_MET;
    else
         NEW_NODE.THE_UPPER :=
              NEW_NODE.THE_LOWER + TEMP.THE_PERIOD - TEMP.THE_MET;
    end if;

  end DETERMINE_THE_UPPER;
```

```
procedure DETERMINE_START_STOP( TEMP          : in OPERATOR;
                                PROC_NUM      : in out INTEGER;
                                NEW_NODE      : in out SCHEDULE_INPUTS;
                                PROC_STOP     : in out PROCESSOR_ARRAY) is
  PROC_FREE: VALUE := NEW_NODE.THE_LOWER;
begin
  NEW_NODE.THE_START := NEW_NODE.THE_LOWER;
  for I in 1..NOP loop
    if PROC_STOP(I) <= NEW_NODE.THE_LOWER then
      if PROC_STOP(I) <= PROC_FREE then
        NEW_NODE.THE_START := NEW_NODE.THE_LOWER;
        PROC_FREE := PROC_STOP(I);
        PROC_NUM := I;
      end if;
    else
      if I = 1 then
        NEW_NODE.THE_START := PROC_STOP(1);
        PROC_NUM := 1;
      elsif PROC_STOP(I) <= NEW_NODE.THE_START then
        NEW_NODE.THE_START := PROC_STOP(I);
        PROC_NUM := I;
      end if;
    end if;
  end loop;
  NEW_NODE.THE_STOP := NEW_NODE.THE_START + TEMP.THE_MET;
  PROC_STOP(PROC_NUM) := NEW_NODE.THE_STOP;
end DETERMINE_START_STOP;

procedure CREATE_ADDL_NODE(NEW_NODE   : in SCHEDULE_INPUTS;
                           TEMP       : in OPERATOR;
                           ADDL_NODE  : in out SCHEDULE_INPUTS) is
begin
  ADDL_NODE.THE_OPERATOR := NEW_NODE.THE_OPERATOR;
  ADDL_NODE.THE_START := TEMP.THE_PERIOD * NEW_NODE.THE_INSTANCE;
                -- store synchronous information
  ADDL_NODE.THE_INSTANCE := NEW_NODE.THE_INSTANCE + 1;
  if not NEW_GRAPH.PIPELINE(NEW_NODE.THE_OPERATOR)
      and then ADDL_NODE.THE_LOWER < NEW_NODE.THE_STOP then
    ADDL_NODE.THE_LOWER := NEW_NODE.THE_STOP;
  end if;
end CREATE_ADDL_NODE;
```

```
procedure TEST_SCHEDULE( HBL       : in INTEGER;
                         AGENDA: in SCHEDULE_ARRAY;
                         COST      : in out INTEGER) is

  V              : SCHEDULE_ARRAY := AGENDA;
  PREVIOUS       : SCHEDULE_INPUTS_LIST.LIST := null;

begin
  COST := 0;
  for I in 1..NOP loop
    while SCHEDULE_INPUTS_LIST.NON_EMPTY(V(I)) loop
      if COST  < (SCHEDULE_INPUTS_LIST.VALUE(V(I)).THE_START
                  - SCHEDULE_INPUTS_LIST.VALUE(V(I)).THE_UPPER) then
        COST := SCHEDULE_INPUTS_LIST.VALUE(V(I)).THE_START
                  - SCHEDULE_INPUTS_LIST.VALUE(V(I)).THE_UPPER;
      end if;
      PREVIOUS := V(I);
      SCHEDULE_INPUTS_LIST.NEXT(V(I));
    end loop;
    if SCHEDULE_INPUTS_LIST.VALUE(PREVIOUS).THE_STOP > HBL and then
          COST < (SCHEDULE_INPUTS_LIST.VALUE(PREVIOUS).THE_STOP - HBL) then
      COST := SCHEDULE_INPUTS_LIST.VALUE(PREVIOUS).THE_STOP - HBL;
    end if;
  end loop;
end TEST_SCHEDULE;

function ANNEAL_FUNCTION( COST_1              : INTEGER;
                          COST_2              : INTEGER;
                          CURRENT_TEMPER      : FLOAT) return FLOAT is

  DELTA_C : FLOAT;

begin
  DELTA_C := (FLOAT(COST_1 - COST_2)/CURRENT_TEMPER);
  if DELTA_C <= 15.0 then
    return EXP(-DELTA_C);
  else
    return 0.0;
  end if;
end ANNEAL_FUNCTION;
```

```
procedure ADJUST_PRECEDENCE( COUNT    : in INTEGER;
                             P_LIST   : in out NODE_LIST.LIST) is

  MOVE_COUNT          : INTEGER := 0;
  OP_NO               : INTEGER;
  PRE_NO              : INTEGER;
  PRECEDENCE_NEW      : BOOLEAN := false;
  WORK_LIST           : NODE_LIST.LIST;
  ADJUST_OP           : NODE_LIST.LIST;
  AHEAD               : NODE_LIST.LIST;

begin
  while not PRECEDENCE_NEW loop
    NODE_LIST.FREE_LIST(WORK_LIST);
    WORK_LIST := P_LIST;
    while NODE_LIST.NON_EMPTY(WORK_LIST) loop        --Move to tail of list
      ADJUST_OP:= WORK_LIST;
      NODE_LIST.NEXT(WORK_LIST);
    end loop;
    MOVE_COUNT := INTEGER(RANDOM_NEXT * FLOAT(COUNT));
    while MOVE_COUNT > 1 loop
      NODE_LIST.PREVIOUS(ADJUST_OP);
      MOVE_COUNT := MOVE_COUNT - 1;
    end loop;
    WORK_LIST := ADJUST_OP;
    OP_NO := NODE_LIST.VALUE(ADJUST_OP);
    NODE_LIST.PREVIOUS(WORK_LIST);
    AHEAD := WORK_LIST;
    while NODE_LIST.NON_EMPTY(AHEAD) and not PRECEDENCE_NEW loop
      PRE_NO := NODE_LIST.VALUE(WORK_LIST);
      if NEW_GRAPH.IS_PARENT(PRE_NO,OP_NO) then
          NODE_LIST.PREVIOUS(ADJUST_OP);
          OP_NO := NODE_LIST.VALUE(ADJUST_OP);
          WORK_LIST := ADJUST_OP;
          NODE_LIST.PREVIOUS(WORK_LIST);
          AHEAD := WORK_LIST;
      else
        while NODE_LIST.NON_EMPTY(WORK_LIST) and
                  not NEW_GRAPH.IS_PARENT(PRE_NO,OP_NO) loop
          NODE_LIST.PREVIOUS(WORK_LIST);
          if NODE_LIST.NON_EMPTY(WORK_LIST) then
```

```
                PRE_NO := NODE_LIST.VALUE(WORK_LIST);
              end if;
            end loop;
            NODE_LIST.REMOVE(OP_NO,P_LIST);
            NODE_LIST.INSERT_NEXT(OP_NO,WORK_LIST);
            PRECEDENCE_NEW := true;
          end if;
        end loop;
      end loop;
    end ADJUST_PRECEDENCE;

end UTILITY_PKG;
```

2. NEW_SCHEDULER_PKG

```
with DATA; use DATA;

package NEW_SCHEDULER_PKG is

  procedure EARLIEST_START( COUNT           : in INTEGER;
                            HBL             : in INTEGER;
                            VALID_SCHEDULE  : in out BOOLEAN;
                            AGENDA          : in out SCHEDULE_ARRAY);

  procedure EARLIEST_DEADLINE(COUNT           : in INTEGER;
                              HBL             : in INTEGER;
                              VALID_SCHEDULE  : in out BOOLEAN;
                              AGENDA          : in out SCHEDULE_ARRAY);

  procedure SIMULATED_ANNEAL(COUNT            : in INTEGER;
                             HBL              : in INTEGER;
                             PRECEDENCE_LIST  : in out NODE_LIST.LIST;
                             AGENDA           : in out SCHEDULE_ARRAY;
                             VALID_SCHEDULE   : in out BOOLEAN);

end NEW_SCHEDULER_PKG;
```

```
with TEXT_IO; use TEXT_IO;
with DATA; use DATA;
with FRONT_END; use FRONT_END;
with PRIORITY_QUEUES;
with UTILITY_PKG; use UTILITY_PKG;
with MATH; use MATH;
with DIAGNOSTICS;

package body NEW_SCHEDULER_PKG is

  package int_io is new TEXT_IO.INTEGER_IO(INTEGER); use int_io;

  procedure EARLIEST_START( COUNT          : in INTEGER;
                            HBL            : in INTEGER;
                            VALID_SCHEDULE : in out BOOLEAN;
                            AGENDA         : in out SCHEDULE_ARRAY) is

    package EST_QUEUES is new PRIORITY_QUEUES(SCHEDULE_INPUTS,LOWERS,"<");

    type IN_ARRAY is array (0..COUNT) of VALUE;

    PROCESSOR_STOP  : PROCESSOR_ARRAY            := (others => 0);
    REV_AGENDA      : SCHEDULE_ARRAY             := (others => null);
    WORK_LIST       : NODE_LIST.LIST             := null;
    PARENT_HEAD     : NODE_LIST.LIST             := null;
    CHILD_HEAD      : NODE_LIST.LIST             := null;
    QUE             : EST_QUEUES.LINK            := null;
    WAIT_LIST       : SCHEDULE_INPUTS_LIST.LIST  := null;
    PARENT_OP       : OPERATOR;
    CHILD_OP        : OPERATOR;
    CHILD_NUM       : INTEGER;
    TEMP_NODE       : SCHEDULE_INPUTS;
    INSERT          : BOOLEAN;
    FINISH          : IN_ARRAY                   := (others=> 0);
        -- store the finishing time of 1st instance of each op
    INSTANCE_NO     : IN_ARRAY                   := (others=> 0);
        -- store the # of instances of each op that have been scheduled
    READY           : IN_ARRAY                   := (others=> 0);
        -- store the lower bound of the 1st instance of each op
```

```
procedure DETERMINE_BEST_LOWER(REV_AGENDA : in SCHEDULE_ARRAY;
                               INST_NO     : in out IN_ARRAY;
                               NEW_NODE    : in out SCHEDULE_INPUTS;
                               WAIT_LIST   : in out SCHEDULE_INPUTS_LIST.LIST;
                               PRI_Q       : in out EST_QUEUES.LINK) is

  NEXT_PROCESSOR  : BOOLEAN;
  INSTANCE_FOUND  : BOOLEAN;
  OLD             : SCHEDULE_INPUTS;
  TEMP_SCHEDULE   : SCHEDULE_ARRAY;
  OP_NO           : INTEGER;
  I,L             : NATURAL;

begin
  OP_NO := NEW_NODE.THE_OPERATOR;
  if not NEW_GRAPH.IS_CHILD(OP_NO,0) then
    NEW_GRAPH.RETURN_PARENT_LIST(PARENT_LIST,OP_NO,PARENT_COUNT);
    LIST_HEAD := PARENT_LIST;
    while NODE_LIST.NON_EMPTY(PARENT_LIST) loop
      PARENT_NUM := NODE_LIST.VALUE(PARENT_LIST);
      PARENT_OP  := NEW_GRAPH.OP_RETURN(PARENT_NUM);
      if (NEW_NODE.THE_START mod PARENT_OP.THE_PERIOD) = 0 then
        I := (NEW_NODE.THE_START / PARENT_OP.THE_PERIOD) + 1 ;
        if I > INST_NO(PARENT_NUM) then
          SCHEDULE_INPUTS_LIST.ADD(NEW_NODE,WAIT_LIST);
          NEW_NODE := EST_QUEUES.READ_BEST_FROM_PRIORITY_QUEUE(PRI_Q);
          EST_QUEUES.REMOVE_BEST_FROM_PRIORITY_QUEUE(PRI_Q);
          DETERMINE_BEST_LOWER(REV_AGENDA,INST_NO,NEW_NODE,WAIT_LIST,PRI_Q);
        else
          TEMP_SCHEDULE := REV_AGENDA;
          L := 1;
          INSTANCE_FOUND := false;
          while L <= NOP and not INSTANCE_FOUND loop
            NEXT_PROCESSOR := false;
            while not (INSTANCE_FOUND or NEXT_PROCESSOR) loop
              if SCHEDULE_INPUTS_LIST.NON_EMPTY(TEMP_SCHEDULE(L)) then
                OLD := SCHEDULE_INPUTS_LIST.VALUE(TEMP_SCHEDULE(L));
                if OLD.THE_STOP > NEW_NODE.THE_LOWER then
                  if OLD.THE_OPERATOR = PARENT_NUM and then
                      OLD.THE_INSTANCE = I then
                    INSTANCE_FOUND := true;
```

```
                    if NEW_NODE.THE_LOWER < OLD.THE_STOP +
                            NEW_GRAPH.LATENCY(PARENT_NUM,OP_NO) then
                      NEW_NODE.THE_LOWER:= OLD.THE_STOP +
                          NEW_GRAPH.LATENCY(PARENT_NUM,OP_NO);
                    end if;
                  else
                    SCHEDULE_INPUTS_LIST.NEXT(TEMP_SCHEDULE(L));
                  end if;
                else
                  NEXT_PROCESSOR := true;
                end if;
              else
                NEXT_PROCESSOR := true;
              end if;
            end loop;
            L := L + 1;
          end loop;
        end if; -- "I > INST_NO"
      end if;
      NODE_LIST.NEXT(PARENT_LIST);
    end loop;
    NODE_LIST.FREE_LIST(LIST_HEAD);
  end if;
end DETERMINE_BEST_LOWER;

begin -- procedure EARLIEST_START
  NODE_LIST.FREE_LIST(PARENT_LIST);
  NODE_LIST.FREE_LIST(CHILD_LIST);
  NEW_GRAPH.RETURN_CHILD_LIST(WORK_LIST,0,CHILD_COUNT);
  LIST_HEAD := WORK_LIST;
  while NODE_LIST.NON_EMPTY(WORK_LIST) loop
    OP_NUM := NODE_LIST.VALUE(WORK_LIST);
    TEMP := NEW_GRAPH.OP_RETURN(OP_NUM);
    NEW_INPUT.THE_OPERATOR := OP_NUM;
    NEW_INPUT.THE_LOWER := 0;
    NEW_INPUT.THE_INSTANCE := 1;
    DETERMINE_THE_UPPER(TEMP,NEW_INPUT);
    DETERMINE_START_STOP(TEMP,PROCESSOR_NUM,NEW_INPUT,PROCESSOR_STOP)
    READY(OP_NUM) := NEW_INPUT.THE_START;
    FINISH(OP_NUM) := NEW_INPUT.THE_STOP;
    INSTANCE_NO(OP_NUM) := 1;
```

```
  SCHEDULE_INPUTS_LIST.ADD(NEW_INPUT,REV_AGENDA(PROCESSOR_NUM));
  if NEW_INPUT.THE_INSTANCE < (HBL / TEMP.THE_PERIOD) then
    ADDL_NODE.THE_LOWER := READY(OP_NUM) + TEMP.THE_PERIOD;
    DETERMINE_THE_UPPER(TEMP,ADDL_NODE);
    CREATE_ADDL_NODE(NEW_INPUT,TEMP,ADDL_NODE);
    EST_QUEUES.INSERT_IN_PRIORITY_QUEUE(ADDL_NODE,ADDL_NODE.THE_LOWER,QUE);
  end if;
  NEW_GRAPH.RETURN_CHILD_LIST(CHILD_LIST,OP_NUM,CHILD_COUNT);
  CHILD_HEAD := CHILD_LIST;
  while NODE_LIST.NON_EMPTY(CHILD_LIST) loop
    CHILD_NUM := NODE_LIST.VALUE(CHILD_LIST);
    INSERT := true;
    NEW_GRAPH.RETURN_PARENT_LIST(PARENT_LIST,CHILD_NUM,PARENT_COUNT;
    PARENT_HEAD := PARENT_LIST;
    FIRST.THE_LOWER := 0;
    while NODE_LIST.NON_EMPTY(PARENT_LIST) loop
      PARENT_NUM := NODE_LIST.VALUE(PARENT_LIST);
      if INSTANCE_NO(PARENT_NUM) = 0 then
        INSERT := false;
        PARENT_LIST := null;
      else
        if FINISH(PARENT_NUM) + NEW_GRAPH.
              LATENCY(PARENT_NUM, CHILD_NUM) > FIRST.THE_LOWER then
          FIRST.THE_LOWER := FINISH(PARENT_NUM) + NEW_GRAPH.
                LATENCY(PARENT_NUM,CHILD_NUM);
        end if;
        NODE_LIST.NEXT(PARENT_LIST);
      end if;
    end loop;
    NODE_LIST.FREE_LIST(PARENT_HEAD);
    if INSERT then
      FIRST.THE_OPERATOR := CHILD_NUM;
      CHILD_OP := NEW_GRAPH.OP_RETURN(CHILD_NUM);
      DETERMINE_THE_UPPER(CHILD_OP,FIRST);
      EST_QUEUES.INSERT_IN_PRIORITY_QUEUE(FIRST,FIRST.THE_LOWER,QUE);
    end if;
    NODE_LIST.NEXT(CHILD_LIST);
  end loop;
  NODE_LIST.FREE_LIST(CHILD_HEAD);
  NODE_LIST.NEXT(WORK_LIST);
end loop;
```

```
NODE_LIST.FREE_LIST(LIST_HEAD);

while EST_QUEUES.NON_EMPTY(QUE) loop
  BEST := EST_QUEUES.READ_BEST_FROM_PRIORITY_QUEUE(QUE);
  EST_QUEUES.REMOVE_BEST_FROM_PRIORITY_QUEUE(QUE);
  if BEST.THE_INSTANCE = 1 then
    OP_NUM := BEST.THE_OPERATOR;
    NEW_INPUT := BEST;
    TEMP := NEW_GRAPH.OP_RETURN(OP_NUM);
    DETERMINE_START_STOP(TEMP,PROCESSOR_NUM,NEW_INPUT,PROCESSOR_STOP);
    READY(OP_NUM) := NEW_INPUT.THE_LOWER;
    FINISH(OP_NUM) := NEW_INPUT.THE_STOP;
    INSTANCE_NO(OP_NUM) := 1;
    SCHEDULE_INPUTS_LIST.ADD(NEW_INPUT,REV_AGENDA(PROCESSOR_NUM));
    NEW_GRAPH.RETURN_CHILD_LIST(CHILD_LIST,OP_NUM,CHILD_COUNT);
    CHILD_HEAD := CHILD_LIST;
    while NODE_LIST.NON_EMPTY(CHILD_LIST) loop
      CHILD_NUM := NODE_LIST.VALUE(CHILD_LIST);
      INSERT := true;
      NEW_GRAPH.RETURN_PARENT_LIST(PARENT_LIST,CHILD_NUM,PARENT_COUNT);
      PARENT_HEAD := PARENT_LIST;
      FIRST.THE_LOWER := 0;
      while NODE_LIST.NON_EMPTY(PARENT_LIST) loop
        PARENT_NUM := NODE_LIST.VALUE(PARENT_LIST);
        if INSTANCE_NO(PARENT_NUM) = 0 then
          INSERT := false;
          PARENT_LIST := null;
        else
          if FINISH(PARENT_NUM) + NEW_GRAPH.LATENCY
                (PARENT_NUM,CHILD_NUM) > FIRST.THE_LOWER then
            FIRST.THE_LOWER := FINISH(PARENT_NUM) + NEW_GRAPH.LATENCY
                  (PARENT_NUM,CHILD_NUM);
          end if;
          NODE_LIST.NEXT(PARENT_LIST);
        end if;
      end loop;
      NODE_LIST.FREE_LIST(PARENT_HEAD);
      if INSERT then
        FIRST.THE_OPERATOR := CHILD_NUM;
        CHILD_OP := NEW_GRAPH.OP_RETURN(CHILD_NUM);
        DETERMINE_THE_UPPER(CHILD_OP,FIRST);
```

```
          EST_QUEUES.INSERT_IN_PRIORITY_QUEUE(FIRST,FIRST.THE_LOWER,QUE);
        end if;
        NODE_LIST.NEXT(CHILD_LIST);
      end loop;
      NODE_LIST.FREE_LIST(CHILD_HEAD);
    else
      DETERMINE_BEST_LOWER(REV_AGENDA,INSTANCE_NO,BEST,WAIT_LIST,QUE);
      OP_NUM := BEST.THE_OPERATOR;
      NEW_INPUT := BEST;
      TEMP := NEW_GRAPH.OP_RETURN(OP_NUM);
      DETERMINE_START_STOP(TEMP,PROCESSOR_NUM,NEW_INPUT,PROCESSOR_STOP);
      SCHEDULE_INPUTS_LIST.ADD(NEW_INPUT,REV_AGENDA(PROCESSOR_NUM));
      INSTANCE_NO(NEW_INPUT.THE_OPERATOR) := NEW_INPUT.THE_INSTANCE;
      while SCHEDULE_INPUTS_LIST.NON_EMPTY(WAIT_LIST) loop
        TEMP_NODE := SCHEDULE_INPUTS_LIST.VALUE(WAIT_LIST);
        EST_QUEUES.INSERT_IN_PRIORITY_QUEUE(TEMP_NODE,TEMP_NODE.THE_LOWER,QUE);
        SCHEDULE_INPUTS_LIST.REMOVE(TEMP_NODE,WAIT_LIST);
      end loop;
    end if;
    if NEW_INPUT.THE_START > NEW_INPUT.THE_UPPER
          or NEW_INPUT.THE_STOP > HBL then
      VALID_SCHEDULE := false;
    end if;
    if NEW_INPUT.THE_INSTANCE < (HBL / TEMP.THE_PERIOD) then
      ADDL_NODE.THE_LOWER := READY(NEW_INPUT.THE_OPERATOR)
                                  + TEMP.THE_PERIOD * NEW_INPUT.THE_INSTANCE;
      DETERMINE_THE_UPPER(TEMP,ADDL_NODE);
      CREATE_ADDL_NODE(NEW_INPUT,TEMP,ADDL_NODE);
      EST_QUEUES.INSERT_IN_PRIORITY_QUEUE
            (ADDL_NODE,ADDL_NODE.THE_LOWER,QUE);
    end if;
  end loop;
  for I in 1..NOP loop
      SCHEDULE_INPUTS_LIST.LIST_REVERSE(REV_AGENDA(I), AGENDA(I));
      SCHEDULE_INPUTS_LIST.FREE_LIST(REV_AGENDA(I));
  end loop;

end EARLIEST_START;
```

```
procedure EARLIEST_DEADLINE( COUNT           : in INTEGER;
                             HBL             : in INTEGER;
                             VALID_SCHEDULE  : in out BOOLEAN;
                             AGENDA          : in out SCHEDULE_ARRAY) is

  package EDL_QUEUES is new PRIORITY_QUEUES(SCHEDULE_INPUTS,UPPERS,"<");

  type IN_ARRAY is array (0..COUNT) of VALUE;

  PROCESSOR_STOP    : PROCESSOR_ARRAY  :=  (others => 0);
  REV_AGENDA        : SCHEDULE_ARRAY   :=  (others => null);
  WORK_LIST         : NODE_LIST.LIST   :=  null;
  PARENT_HEAD       : NODE_LIST.LIST   :=  null;
  CHILD_HEAD        : NODE_LIST.LIST   :=  null;
  WAIT_LIST         : SCHEDULE_INPUTS_LIST.LIST := null;
  PARENT_OP         : OPERATOR;
  CHILD_OP          : OPERATOR;
  CHILD_NUM         : INTEGER;
  QUE               : EDL_QUEUES.LINK  :=  null;
  INSERT            : BOOLEAN;
  TEMP_NODE         : SCHEDULE_INPUTS;
  FINISH            : IN_ARRAY         :=  (others=> 0);
  INSTANCE_NO       : IN_ARRAY         :=  (others=> 0);
  READY             : IN_ARRAY         :=  (others=> 0);

  procedure DETERMINE_BEST_LOWER(REV_AGENDA:   in SCHEDULE_ARRAY;
                                 INST_NO      : in out IN_ARRAY;
                                 NEW_NODE     : in out SCHEDULE_INPUTS;
                                 WAIT_LIST    : in out SCHEDULE_INPUTS_LT.LIST;
                                 PRI_Q        : in out EDL_QUEUES.LINK) is

    NEXT_PROCESSOR  : BOOLEAN;
    INSTANCE_FOUND  : BOOLEAN;
    OLD             : SCHEDULE_INPUTS;
    TEMP_SCHEDULE   : SCHEDULE_ARRAY;
    OP_NO           : INTEGER;
    I,L             : NATURAL;

  begin
    OP_NO := NEW_NODE.THE_OPERATOR;
    if not NEW_GRAPH.IS_CHILD(OP_NO,0) then
```

```
NEW_GRAPH.RETURN_PARENT_LIST(PARENT_LIST,OP_NO,PARENT_COUNT);
LIST_HEAD := PARENT_LIST;
while NODE_LIST.NON_EMPTY(PARENT_LIST) loop
  PARENT_NUM :=   NODE_LIST.VALUE(PARENT_LIST);
  PARENT_OP    :=   NEW_GRAPH.OP_RETURN(PARENT_NUM);
  if (NEW_NODE.THE_START mod PARENT_OP.THE_PERIOD) = 0 then
    I := (NEW_NODE.THE_START / PARENT_OP.THE_PERIOD) + 1;
    if I > INST_NO(PARENT_NUM) then
      SCHEDULE_INPUTS_LIST.ADD(NEW_NODE,WAIT_LIST);
      NEW_NODE := EDL_QUEUES.READ_BEST_FROM_PRIORITY_QUEUE(PRI_Q);
      EDL_QUEUES.REMOVE_BEST_FROM_PRIORITY_QUEUE(PRI_Q);
      DETERMINE_BEST_LOWER
            (REV_AGENDA,INST_NO,NEW_NODE,WAIT_LIST,PRI_Q);
    else
      TEMP_SCHEDULE := REV_AGENDA;
      L := 1;
      INSTANCE_FOUND := false;
      while L <= NOP and not INSTANCE_FOUND loop
        NEXT_PROCESSOR := false;
        while not (INSTANCE_FOUND or NEXT_PROCESSOR) loop
          if SCHEDULE_INPUTS_LIST.NON_EMPTY(TEMP_SCHEDULE(L)) then
            OLD := SCHEDULE_INPUTS_LIST.VALUE(TEMP_SCHEDULE(L));
            if OLD.THE_STOP > NEW_NODE.THE_LOWER then
              if OLD.THE_OPERATOR = PARENT_NUM and then
                  OLD.THE_INSTANCE = I then
                INSTANCE_FOUND := true;
                if NEW_NODE.THE_LOWER < OLD.THE_STOP
                      + NEW_GRAPH.LATENCY(PARENT_NUM,OP_NO) then
                  NEW_NODE.THE_LOWER:= OLD.THE_STOP
                      + NEW_GRAPH.LATENCY(PARENT_NUM,OP_NO);
                end if;
              else
                SCHEDULE_INPUTS_LIST.NEXT(TEMP_SCHEDULE(L));
              end if;
            else
              NEXT_PROCESSOR := true;
            end if;
          else
            NEXT_PROCESSOR := true;
          end if;
        end loop;
```

```
            L := L + 1;
          end loop;
        end if; -- "I > INST_NO"
      end if;
      NODE_LIST.NEXT(PARENT_LIST);
    end loop;
    NODE_LIST.FREE_LIST(LIST_HEAD);
  end if;

end DETERMINE_BEST_LOWER;

begin -- procedure EARLIEST_DEADLINE
  NODE_LIST.FREE_LIST(PARENT_LIST);
  NODE_LIST.FREE_LIST(CHILD_LIST);

  NEW_GRAPH.RETURN_CHILD_LIST(WORK_LIST,0,CHILD_COUNT);
  LIST_HEAD := WORK_LIST;
  while NODE_LIST.NON_EMPTY(WORK_LIST) loop
    OP_NUM := NODE_LIST.VALUE(WORK_LIST);
    TEMP := NEW_GRAPH.OP_RETURN(OP_NUM);
    FIRST.THE_OPERATOR := OP_NUM;
    FIRST.THE_LOWER := 0;
    DETERMINE_THE_UPPER(TEMP,FIRST);
    EDL_QUEUES.INSERT_IN_PRIORITY_QUEUE(FIRST,FIRST.THE_UPPER,QUE);
    NODE_LIST.NEXT(WORK_LIST);
  end loop;
  NODE_LIST.FREE_LIST(LIST_HEAD);

  while EDL_QUEUES.NON_EMPTY(QUE) loop
    BEST := EDL_QUEUES.READ_BEST_FROM_PRIORITY_QUEUE(QUE);
    EDL_QUEUES.REMOVE_BEST_FROM_PRIORITY_QUEUE(QUE);

    if BEST.THE_INSTANCE = 1 then
      OP_NUM := BEST.THE_OPERATOR;
      NEW_INPUT := BEST;
      TEMP := NEW_GRAPH.OP_RETURN(OP_NUM);
      DETERMINE_START_STOP
            (TEMP,PROCESSOR_NUM,NEW_INPUT,PROCESSOR_STOP);
      if NEW_GRAPH.IS_CHILD(OP_NUM,0) then
        READY(OP_NUM) := NEW_INPUT.THE_START;
      else
```

```
    READY(OP_NUM) := NEW_INPUT.THE_LOWER;
  end if;
  FINISH(OP_NUM) := NEW_INPUT.THE_STOP;
  INSTANCE_NO(OP_NUM) := 1;
  SCHEDULE_INPUTS_LIST.ADD(NEW_INPUT,REV_AGENDA(PROCESSOR_NUM));

  NEW_GRAPH.RETURN_CHILD_LIST(CHILD_LIST,OP_NUM,CHILD_COUNT);
  CHILD_HEAD := CHILD_LIST;
  while NODE_LIST.NON_EMPTY(CHILD_LIST) loop
    CHILD_NUM := NODE_LIST.VALUE(CHILD_LIST);
    INSERT := true;
    NEW_GRAPH.RETURN_PARENT_LIST
          (PARENT_LIST,CHILD_NUM,PARENT_COUNT);
    PARENT_HEAD := PARENT_LIST;
    FIRST.THE_LOWER := 0;
    while NODE_LIST.NON_EMPTY(PARENT_LIST) loop
      PARENT_NUM := NODE_LIST.VALUE(PARENT_LIST);
      if INSTANCE_NO(PARENT_NUM) = 0 then
        INSERT := false;
        PARENT_LIST := null;
      else
        if FIRST.THE_LOWER < FINISH(PARENT_NUM)
              + NEW_GRAPH.LATENCY(PARENT_NUM,CHILD_NUM) then
          FIRST.THE_LOWER := FINISH(PARENT_NUM)
                + NEW_GRAPH.LATENCY(PARENT_NUM,CHILD_NUM);
        end if;
        NODE_LIST.NEXT(PARENT_LIST);
      end if;
    end loop;
    NODE_LIST.FREE_LIST(PARENT_HEAD);

    if INSERT then
      FIRST.THE_OPERATOR := CHILD_NUM;
      CHILD_OP := NEW_GRAPH.OP_RETURN(CHILD_NUM);
      DETERMINE_THE_UPPER(CHILD_OP,FIRST);
      EDL_QUEUES.INSERT_IN_PRIORITY_QUEUE(FIRST,FIRST.THE_UPPER,QUE);
    end if;
    NODE_LIST.NEXT(CHILD_LIST);
  end loop;
  NODE_LIST.FREE_LIST(CHILD_HEAD);
else
```

```
      DETERMINE_BEST_LOWER(REV_AGENDA,INSTANCE_NO,BEST,WAIT_LIST,QUE);
      OP_NUM := BEST.THE_OPERATOR;
      NEW_INPUT := BEST;
      TEMP := NEW_GRAPH.OP_RETURN(OP_NUM);
      DETERMINE_START_STOP(TEMP,PROCESSOR_NUM,NEW_INPUT,PROCESSOR_STOP);
      SCHEDULE_INPUTS_LIST.ADD(NEW_INPUT,REV_AGENDA(PROCESSOR_NUM));
      INSTANCE_NO(NEW_INPUT.THE_OPERATOR) := NEW_INPUT.THE_INSTANCE;
      while SCHEDULE_INPUTS_LIST.NON_EMPTY(WAIT_LIST) loop
        TEMP_NODE := SCHEDULE_INPUTS_LIST.VALUE(WAIT_LIST);
        EDL_QUEUES.INSERT_IN_PRIORITY_QUEUE
                (TEMP_NODE,TEMP_NODE.THE_UPPER,QUE);
        SCHEDULE_INPUTS_LIST.REMOVE(TEMP_NODE,WAIT_LIST);
      end loop;
    end if;

    if NEW_INPUT.THE_START > NEW_INPUT.THE_UPPER
        or NEW_INPUT.THE_STOP > HBL then
      VALID_SCHEDULE := false;
    end if;

    if NEW_INPUT.THE_INSTANCE < (HBL / TEMP.THE_PERIOD) then
      ADDL_NODE.THE_LOWER := READY(NEW_INPUT.THE_OPERATOR) +
                    TEMP.THE_PERIOD * NEW_INPUT.THE_INSTANCE;
      DETERMINE_THE_UPPER(TEMP,ADDL_NODE);
      CREATE_ADDL_NODE(NEW_INPUT,TEMP,ADDL_NODE);
      EDL_QUEUES.INSERT_IN_PRIORITY_QUEUE
                    (ADDL_NODE,ADDL_NODE.THE_UPPER,QUE);
    end if;

  end loop;

  for I in 1..NOP loop
      SCHEDULE_INPUTS_LIST.LIST_REVERSE(REV_AGENDA(I), AGENDA(I));
      SCHEDULE_INPUTS_LIST.FREE_LIST(REV_AGENDA(I));
  end loop;

end EARLIEST_DEADLINE;
```

```
procedure SIMULATED_ANNEAL(  COUNT           : in INTEGER;
                             HBL             : in INTEGER;
                             PRECEDENCE_LIST : in out NODE_LIST.LIST;
                             AGENDA          : in out SCHEDULE_ARRAY;
                             VALID_SCHEDULE  : in out BOOLEAN) is

  package float_io is new TEXT_IO.FLOAT_IO(FLOAT); use float_io;
  package PRIORITY_Q is new PRIORITY_QUEUES(SCHEDULE_INPUTS,LOWERS,"<");
  use PRIORITY_Q;

  type IN_ARRAY is array (0..COUNT) of VALUE;
  PROCESSOR_STOP  : PROCESSOR_ARRAY;
  REV_AGENDA      : SCHEDULE_ARRAY;
  INSTANCE_NO     : IN_ARRAY;
  READY           : IN_ARRAY;
  P_LIST          : NODE_LIST.LIST := null;
  ADDL_LIST       : SCHEDULE_INPUTS_LIST.LIST;
  A_LIST          : SCHEDULE_INPUTS_LIST.LIST  :=  null;
  HEAD_A          : SCHEDULE_INPUTS_LIST.LIST  :=  null;
  COST            : INTEGER;
  QUE             : PRIORITY_Q.LINK := null;
```

```
procedure SCHEDULE_1st_INSTANCES( HBL            : in     INTEGER;
                                  READY          : in out IN_ARRAY;
                                  P_LIST         : in out NODE_LIST.LIST;
                                  PROC_STOP      : in out PROCESSOR_ARRAY;
                                  REV_AGENDA     : in out SCHEDULE_ARRAY;
                                  A_LIST: in out SCHEDULE_INPUTS_LIST.LIST) is

  FINISH          : IN_ARRAY := (others => 0);
  WORK_LIST       : NODE_LIST.LIST;
  PARENT_HEAD     : NODE_LIST.LIST;

begin
  NODE_LIST.FREE_LIST(PARENT_LIST);
  NODE_LIST.FREE_LIST(CHILD_LIST);
  SCHEDULE_INPUTS_LIST.FREE_LIST(A_LIST);
  REV_AGENDA := (others=> nuII);
  PROC_STOP := (others=> 0);
  READY := (others => 0);
  NODE_LIST.REMOVE(0,P_LIST);
  NODE_LIST.DUPLICATE(P_LIST,WORK_LIST);
  NEW_GRAPH.RETURN_CHILD_LIST(CHILD_LIST,0,CHILD_COUNT);
  LIST_HEAD := CHILD_LIST;
  while NODE_LIST.NON_EMPTY(CHILD_LIST) loop
    OP_NUM := NODE_LIST.VALUE(CHILD_LIST);
    TEMP := NEW_GRAPH.OP_RETURN(OP_NUM);
    NEW_INPUT.THE_OPERATOR := OP_NUM;
    NEW_INPUT.THE_LOWER := 0;
    NEW_INPUT.THE_INSTANCE := 1;
    DETERMINE_THE_UPPER(TEMP,NEW_INPUT);
    DETERMINE_START_STOP(TEMP,PROCESSOR_NUM,NEW_INPUT,PROC_STOP);
    READY(OP_NUM) := NEW_INPUT.THE_START;
    FINISH(OP_NUM) := NEW_INPUT.THE_STOP;
    SCHEDULE_INPUTS_LIST.ADD(NEW_INPUT,REV_AGENDA(PROCESSOR_NUM));
    NODE_LIST.REMOVE(OP_NUM,WORK_LIST);
    if NEW_INPUT.THE_INSTANCE = (HBL / TEMP.THE_PERIOD) then
      NODE_LIST.REMOVE(OP_NUM,P_LIST);
    else
      ADDL_NODE.THE_LOWER := READY(OP_NUM) + TEMP.THE_PERIOD;
      DETERMINE_THE_UPPER(TEMP,ADDL_NODE);
      CREATE_ADDL_NODE(NEW_INPUT,TEMP,ADDL_NODE);
      SCHEDULE_INPUTS_LIST.ADD(ADDL_NODE,A_LIST);
```

```
    end if;
    NODE_LIST.NEXT(CHILD_LIST);
  end loop;
  NODE_LIST.FREE_LIST(LIST_HEAD);
  LIST_HEAD:= WORK_LIST;
  while NODE_LIST.NON_EMPTY(WORK_LIST) loop
    OP_NUM := NODE_LIST.VALUE(WORK_LIST);
    TEMP := NEW_GRAPH.OP_RETURN(OP_NUM);
    NEW_INPUT.THE_OPERATOR := OP_NUM;
    NEW_INPUT.THE_LOWER := 0;
    NEW_GRAPH.RETURN_PARENT_LIST(PARENT_LIST,OP_NUM,PARENT_COUNT);
    PARENT_HEAD := PARENT_LIST;
    while NODE_LIST.NON_EMPTY(PARENT_LIST) loop
      PARENT_NUM := NODE_LIST.VALUE(PARENT_LIST);
      PARENT_OP  := NEW_GRAPH.OP_RETURN(PARENT_NUM);
      if NEW_INPUT.THE_LOWER < FINISH(PARENT_NUM)
                  + NEW_GRAPH.LATENCY(PARENT_NUM,OP_NUM) then
        NEW_INPUT.THE_LOWER := FINISH(PARENT_NUM)
            + NEW_GRAPH.LATENCY(PARENT_NUM,OP_NUM);
      end if;
      NODE_LIST.NEXT(PARENT_LIST);
    end loop;
    NODE_LIST.FREE_LIST(PARENT_HEAD);
    DETERMINE_THE_UPPER(TEMP,NEW_INPUT);
    DETERMINE_START_STOP(TEMP,PROCESSOR_NUM,NEW_INPUT,PROC_STOP);
    READY(OP_NUM) := NEW_INPUT.THE_LOWER;
    FINISH(OP_NUM) := NEW_INPUT.THE_STOP;
    SCHEDULE_INPUTS_LIST.ADD(NEW_INPUT,REV_AGENDA(PROCESSOR_NUM));
    if NEW_INPUT.THE_INSTANCE = (HBL / TEMP.THE_PERIOD) then
      NODE_LIST.REMOVE(OP_NUM,P_LIST);
    else
      ADDL_NODE.THE_LOWER := READY(OP_NUM) + TEMP.THE_PERIOD;
      DETERMINE_THE_UPPER(TEMP,ADDL_NODE);
      CREATE_ADDL_NODE(NEW_INPUT,TEMP,ADDL_NODE);
      SCHEDULE_INPUTS_LIST.ADD(ADDL_NODE,A_LIST);
    end if;
    NODE_LIST.NEXT(WORK_LIST);
  end loop;
  NODE_LIST.FREE_LIST(LIST_HEAD);

end SCHEDULE_1st_INSTANCES;
```

```
procedure SCHEDULE_REST_OF_BLOCK( HBL          : in INTEGER;
                                  P_LIST       : in out NODE_LIST.LIST;
                                  A_LIST       : in out SCHEDULE_INPUTS_LIST.LIST;
                                  REV_AGENDA: in out SCHEDULE_ARRAY;
                                  PROC_STOP    : in out PROCESSOR_ARRAY;
                                  INST_NO      : in out IN_ARRAY;
                                  READY        : in IN_ARRAY) is

  WORK_LIST : NODE_LIST.LIST;
  HEAD_A    : SCHEDULE_INPUTS_LIST.LIST;
  OP_FOUND  : BOOLEAN;
  AWAIT     : BOOLEAN;

procedure DETERMINE_THE_LOWER( REV_AGENDA : in SCHEDULE_ARRAY;
                               AWAIT      : out BOOLEAN;
                               INST_NO    : in out IN_ARRAY;
                               NEW_NODE   : in out SCHEDULE_INPUTS) is

    PARENT_FOUND     : BOOLEAN;
    OLD              : SCHEDULE_INPUTS;
    TEMP_SCHEDULE    : SCHEDULE_ARRAY;
    OP_NO            : INTEGER;
    J,M              : INTEGER;

  begin

    OP_NO := NEW_NODE.THE_OPERATOR;
    NEW_GRAPH.RETURN_PARENT_LIST(PARENT_LIST,OP_NO,PARENT_COUNT);
    if NODE_LIST.VALUE(PARENT_LIST) = 0 then
      NODE_LIST.REMOVE(0,PARENT_LIST);
    end if;
    LIST_HEAD := PARENT_LIST;
    while NODE_LIST.NON_EMPTY(PARENT_LIST) loop
      PARENT_NUM := NODE_LIST.VALUE(PARENT_LIST);
      PARENT_OP  := NEW_GRAPH.OP_RETURN(PARENT_NUM);
      if (NEW_NODE.THE_START mod PARENT_OP.THE_PERIOD) = 0 then
        J := (NEW_NODE.THE_START / PARENT_OP.THE_PERIOD) + 1;
        AWAIT := false;
        if J > INST_NO(PARENT_NUM) then
          AWAIT := true;
          NODE_LIST.FREE_LIST(LIST_HEAD);
```

```
          exit;
        else
          TEMP_SCHEDULE := REV_AGENDA;
          PARENT_FOUND := false;
          for M in 1..NOP loop
            while SCHEDULE_INPUTS_LIST.NON_EMPTY(TEMP_SCHEDULE(M))
                and not PARENT_FOUND loop
              OLD := SCHEDULE_INPUTS_LIST.VALUE(TEMP_SCHEDULE(M));
              if OLD.THE_STOP > NEW_NODE.THE_LOWER then
                if OLD.THE_OPERATOR = PARENT_NUM
                    and then OLD.THE_INSTANCE = J then
                  PARENT_FOUND := true;
                  TEMP_SCHEDULE(M) := null;
                  if NEW_NODE.THE_LOWER < OLD.THE_STOP
                      NEW_GRAPH.LATENCY(PARENT_NUM,OP_NO) then
                    NEW_NODE.THE_LOWER := OLD.THE_STOP
                        + NEW_GRAPH.LATENCY(PARENT_NUM,OP_NO);
                  end if;
                else
                  SCHEDULE_INPUTS_LIST.NEXT(TEMP_SCHEDULE(M));
                end if;
              else
                TEMP_SCHEDULE(M) := null;
              end if;
            end loop;
            if PARENT_FOUND then
              exit;
            end if;
          end loop;
        end if; -- "if J > "
      end if;
      NODE_LIST.NEXT(PARENT_LIST);
    end loop;
    NODE_LIST.FREE_LIST(LIST_HEAD);

end DETERMINE_THE_LOWER;
```

```
begin --  schedule the rest of the block

  INST_NO := (others=> 1);
  while NODE_LIST.NON_EMPTY(P_LIST) loop
    NODE_LIST.DUPLICATE(P_LIST,WORK_LIST);
    LIST_HEAD := WORK_LIST;
    while NODE_LIST.NON_EMPTY(WORK_LIST) loop
      OP_NUM := NODE_LIST.VALUE(WORK_LIST);
      TEMP:= NEW_GRAPH.OP_RETURN(OP_NUM);
      OP_FOUND:= false;
      HEAD_A := A_LIST;
      while SCHEDULE_INPUTS_LIST.VALUE(A_LIST).THE_OPERATOR /= OP_NUM loop
            SCHEDULE_INPUTS_LIST.NEXT(A_LIST);
      end loop;
      BEST := SCHEDULE_INPUTS_LIST.VALUE(A_LIST);
      NEW_INPUT := BEST;
      DETERMINE_THE_LOWER(REV_AGENDA,AWAIT,INST_NO,NEW_INPUT);
      if not AWAIT then
        DETERMINE_START_STOP(TEMP,PROCESSOR_NUM,NEW_INPUT.PROC_STOP);
        SCHEDULE_INPUTS_LIST.ADD(NEW_INPUT,REV_AGENDA(PROCESSOR_NUM));
        INST_NO(OP_NUM) := NEW_INPUT.THE_INSTANCE;
        if NEW_INPUT.THE_INSTANCE = (HBL / TEMP.THE_PERIOD) then
          NODE_LIST.REMOVE(OP_NUM.P_LIST);
        else
          ADDL_NODE.THE_LOWER := READY(OP_NUM)
                + TEMP.THE_PERIOD * NEW_INPUT.THE_INSTANCE;
          DETERMINE_THE_UPPER(TEMP,ADDL_NODE);
          CREATE_ADDL_NODE(NEW_INPUT,TEMP,ADDL_NODE);
          SCHEDULE_INPUTS_LIST.INSERT_NEXT(ADDL_NODE,A_LIST);
        end if;
        A_LIST := HEAD_A;
        SCHEDULE_INPUTS_LIST.REMOVE(BEST,A_LIST);
      else
        A_LIST := HEAD_A;
      end if;
      NODE_LIST.NEXT(WORK_LIST);
    end loop;
    NODE_LIST.FREE_LIST(LIST_HEAD);
  end loop;

end SCHEDULE_REST_OF_BLOCK;
```

```
procedure ANNEAL_PROCESS( HBL              : in INTEGER;
                          AGENDA           : in out SCHEDULE_ARRAY;
                          SOLUTION         : in out BOOLEAN;
                          PENALTY_COST     : in out INTEGER;
                          INST_NO          : in out IN_ARRAY;
                          PREC_LIST        : in out NODE_LIST.LIST) is
  BEST_AGENDA        : SCHEDULE_ARRAY;
  TEMP_AGENDA        : SCHEDULE_ARRAY;
  TEMPERATURE        : FLOAT;
  BEST_COST          : INTEGER;
  TEMP_COST          : INTEGER;
  TRIAL_NUM          : INTEGER    :=  100;
  ACCEPT_NUM         : INTEGER    :=  35;
  FREEZE             : FLOAT      :=  1.0;
  COOLING_FACTOR     : FLOAT      :=  0.95;
  TRIAL_COUNT        : INTEGER;
  ACCEPT_COUNT       : INTEGER;
  P_LIST_NEW         : BOOLEAN    :=  false;
  P_LIST             : NODE_LIST.LIST;
  AP_FOUND           : BOOLEAN;
  REARRANGE_P        : BOOLEAN;
  TEMP_LOWER         : INTEGER;
  OLD_LOWER          : INTEGER;
  V                  : SCHEDULE_ARRAY;

  procedure DETERMINE_NEW_LOWER( TEMP          : in OPERATOR;
                                 TEMP_AGENDA   : in SCHEDULE_ARRAY;
                                 NEW_NODE      : in out SCHEDULE_INPUTS)is
    NEXT_PROC   : BOOLEAN;
    INS_FOUND   : BOOLEAN;
    INST, I     : INTEGER;
    OP_NO       : INTEGER;
    W           : SCHEDULE_ARRAY := TEMP_AGENDA;

  begin
    OP_NO := NEW_NODE. THE_OPERATOR;
    if TEMP.THE_WITHIN /= 0 then
      NEW_NODE.THE_LOWER := NEW_NODE.THE_UPPER + TEMP.THE_MET-TEMP.THE_WITHIN;
    else
      NEW_NODE.THE_LOWER := NEW_NODE.THE_UPPER + TEMP.THE_MET - TEMP.THE_PERIOD
    end if;
```

```
if not NEW_GRAPH.PIPELINE(OP_NO) then
  INS_FOUND := false;
  for I in 1..NOP loop
    NEXT_PROC := false;
    while SCHEDULE_INPUTS_LIST.NON_EMPTY(W(I)) and not NEXT_PROC loop
      if SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_OPERATOR = OP_NO then
        if SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_INSTANCE
            >= NEW_NODE.THE_INSTANCE - 1 then
          NEXT_PROC := true;
          if SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_INSTANCE
              = NEW_NODE.THE_INSTANCE - 1 then
            INS_FOUND := true;
            if SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_STOP
                > NEW_NODE.THE_LOWER then
              NEW_NODE.THE_LOWER :=
                  SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_STOP;
            end if;
          end if;
        else
          SCHEDULE_INPUTS_LIST.NEXT(W(I));
        end if;
      else
        if SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_START >= NEW_NODE.THE_START then
          NEXT_PROC := true;
        else
          SCHEDULE_INPUTS_LIST.NEXT(W(I));
        end if;
      end if;
    end loop;
    if INS_FOUND then
      exit;
    end if;
  end loop;
end if; --" not PIPELINE"
NEW_GRAPH.RETURN_PARENT_LIST(PARENT_LIST,OP_NO,PARENT_COUNT);
LIST_HEAD := PARENT_LIST;
if NODE_LIST.VALUE(PARENT_LIST) = 0 then
  NODE_LIST.NEXT(PARENT_LIST);
end if;
while NODE_LIST.NON_EMPTY(PARENT_LIST) loop
  PARENT_NUM    :=   NODE_LIST.VALUE(PARENT_LIST);
```

```
    PARENT_OP       :=  NEW_GRAPH.OP_RETURN(PARENT_NUM);
    if ((NEW_NODE.THE_INSTANCE - 1)*(TEMP.THE_PERIOD))
          mod PARENT_OP.THE_PERIOD = 0 then
      INST := (((NEW_NODE.THE_INSTANCE - 1) * (TEMP.THE_PERIOD))
                    / PARENT_OP.THE_PERIOD) + 1;
      INS_FOUND := false;
      W := TEMP_AGENDA;
      for I in 1..NOP loop
        NEXT_PROC := false;
        while SCHEDULE_INPUTS_LIST.NON_EMPTY(W(I)) and not NEXT_PROC loop
          if SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_OPERATOR = PARENT_NUM then
            if SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_INSTANCE >= INST then
              NEXT_PROC := true;
              if SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_INSTANCE = INST then
                INS_FOUND := true;
                if SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_STOP+NEW_GRAPH.LATENCY(PARENT_NUM,OP_NO)
                      > NEW_NODE.THE_LOWER then
                  NEW_NODE.THE_LOWER:= SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_STOP
                      + NEW_GRAPH.LATENCY(PARENT_NUM,OP_NO);
                end if;
              end if;
            else
              SCHEDULE_INPUTS_LIST.NEXT(W(I));
            end if;
          else
            if SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_START >= NEW_NODE.THE_START
            then NEXT_PROC := true;
            else
              SCHEDULE_INPUTS_LIST.NEXT(W(I));
            end if;
          end if;
        end loop;
        if INS_FOUND then
          exit;
        end if;
      end loop;
    end if;
    NODE_LIST.NEXT(PARENT_LIST);
  end loop;
  NODE_LIST.FREE_LIST(LIST_HEAD);
end DETERMINE_NEW_LOWER;
```

```
procedure ADJUST_IT(TEMP          : in OPERATOR;
                    REARRANGE     : in out BOOLEAN;
                    NEW_NODE      : in out SCHEDULE_INPUTS;
                    TEMP_AGENDA   : in out SCHEDULE_ARRAY) is
  DIFF, I                         : INTEGER;
  NEW_PROCESSOR                   : INTEGER;
  STOP_TIME                       : INTEGER;
  W                               : SCHEDULE_ARRAY := TEMP_AGENDA;

begin
  DIFF := -1;
  for I in 1..NOP loop
    while SCHEDULE_INPUTS_LIST.NON_EMPTY(W(I)) loop
      if SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_STOP > NEW_NODE.THE_LOWER then
        if (SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_START - NEW_NODE.THE_STOP) > DIFF then
          DIFF:=(SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_START-NEW_NODE.THE_STOP);
          NEW_PROCESSOR := I;
        end if;
        exit;
      else
        SCHEDULE_INPUTS_LIST.NEXT(W(I));
      end if;
    end loop;
  end loop
  if DIFF < 0 then
    NEW_NODE.THE_START := NEW_NODE.THE_UPPER + 10;
    for I in 1..NOP loop
      if SCHEDULE_INPUTS_LIST.NON_EMPTY(W(I)) then
        STOP_TIME := SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_STOP;
        SCHEDULE_INPUTS_LIST.NEXT(W(I));
        while SCHEDULE_INPUTS_LIST.NON_EMPTY(W(I)) loop
          if SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_START - STOP_TIME < TEMP.THE_MET then
            STOP_TIME := SCHEDULE_INPUTS_LIST.VALUE(W(I)).THE_STOP;
            SCHEDULE_INPUTS_LIST.NEXT(W(I));
          else
            exit;
          end if;
        end loop;
        if STOP_TIME < NEW_NODE.THE_START then
          NEW_NODE.THE_START := STOP_TIME;
          NEW_PROCESSOR := I;
```

```
            end if;
          end if;
        end loop;
      end if;
      if NEW_NODE.THE_START > NEW_NODE.THE_UPPER then
        REARRANGE := true;
      else
        NEW_NODE.THE_STOP  := NEW_NODE.THE_START + TEMP.THE_MET;
        SCHEDULE_INPUTS_LIST.PREVIOUS(W(NEW_PROCESSOR));
        SCHEDULE_INPUTS_LIST.INSERT_NEXT(NEW_NODE,W(NEW_PROCESSOR));
      end if;

    end ADJUST_IT;

  begin-- ANNEAL PROCESS!
    for I in 1..NOP loop
      SCHEDULE_INPUTS_LIST.DUPLICATE(AGENDA(I), BEST_AGENDA(I));
      SCHEDULE_INPUTS_LIST.DUPLICATE(AGENDA(I), TEMP_AGENDA(I));
    end loop;
    TEMPERATURE := 0.9 * FLOAT(PENALTY_COST);
    BEST_COST := PENALTY_COST;
    while not SOLUTION and TEMPERATURE > FREEZE loop
      ACCEPT_COUNT := 0;
      TRIAL_COUNT := 0;
      while not SOLUTION and ACCEPT_COUNT < ACCEPT_NUM and TRIAL_COUNT < TRIAL_NUM loop
        REARRANGE_P := false;
        QUE := INITIALIZE_PRIORITY_QUEUE;
        V := TEMP_AGENDA;
        for I in 1..NOP loop
          AP_FOUND := false;
          while SCHEDULE_INPUTS_LIST.NON_EMPTY(V(I)) and not AP_FOUND loop
            if SCHEDULE_INPUTS_LIST.VALUE(V(I)).THE_START
                > SCHEDULE_INPUTS_LIST.VALUE(V(I)).THE_UPPER then
              TEMP_LOWER := SCHEDULE_INPUTS_LIST.VALUE(V(I)).THE_LOWER;
              AP_FOUND := true;
              INSERT_IN_PRIORITY_QUEUE
                  (SCHEDULE_INPUTS_LIST.VALUE(V(I)),TEMP_LOWER,QUE);
            else
              SCHEDULE_INPUTS_LIST.NEXT(V(I));
            end if;
          end loop;
```

```
end loop;

while PRIORITY_Q.NON_EMPTY(QUE) loop
  BEST := READ_BEST_FROM_PRIORITY_QUEUE(QUE);
  OP_NUM := BEST.THE_OPERATOR;
  for I in 1..NOP loop
    if SCHEDULE_INPUTS_LIST.NON_EMPTY(V(I)) and then
          SCHEDULE_INPUTS_LIST.VALUE(V(I)) = BEST then
      PROCESSOR_NUM := I;
      exit;
    end if;
  end loop;
  NEW_INPUT := BEST;
  TEMP := NEW_GRAPH.OP_RETURN(OP_NUM);
  DETERMINE_NEW_LOWER(TEMP,TEMP_AGENDA,NEW_INPUT);
  if NEW_INPUT.THE_LOWER > NEW_INPUT.THE_UPPER then
    REARRANGE_P := true;
    exit;
  end if;
  if NEW_INPUT.THE_LOWER < NEW_INPUT.THE_START then
    NEW_INPUT.THE_START := NEW_INPUT.THE_LOWER;
    NEW_INPUT.THE_STOP := NEW_INPUT.THE_START + TEMP.THE_MET;
    ADJUST_IT(TEMP,REARRANGE_P,NEW_INPUT,TEMP_AGENDA);
    if REARRANGE_P then
      exit;
    else
      SCHEDULE_INPUTS_LIST.NEXT(V(PROCESSOR_NUM));
      SCHEDULE_INPUTS_LIST.REMOVE(BEST,TEMP_AGENDA(PROCESSOR_NUM));
      REMOVE_BEST_FROM_PRIORITY_QUEUE(QUE);
      if SCHEDULE_INPUTS_LIST.NON_EMPTY(V(PROCESSOR_NUM)) then
        TEMP_LOWER:=SCHEDULE_INPUTS_LIST.VALUE(V(PROCESSOR)).THE_LOWER;
        INSERT_IN_PRIORITY_QUEUE
            (SCHEDULE_INPUTS_LIST.VALUE(V(PROCESSOR_NUM)),TEMP_LOWER,QUE);
      end if;
    end if;
  else
    REMOVE_BEST_FROM_PRIORITY_QUEUE(QUE);
    SCHEDULE_INPUTS_LIST.NEXT(V(PROCESSOR_NUM));
    if SCHEDULE_INPUTS_LIST.NON_EMPTY(V(PROCESSOR_NUM)) then
      TEMP_LOWER
          := SCHEDULE_INPUTS_LIST.VALUE(V(PROCESSOR_NUM)).THE_LOWER;
```

```
        INSERT_IN_PRIORITY_QUEUE
        (SCHEDULE_INPUTS_LIST.VALUE(V(PROCESSOR_NUM)),TEMP_LOWER,QUE);
      end if;
    end if;
  end loop;

  if not REARRANGE_P then
    QUE := INITIALIZE_PRIORITY_QUEUE;
    V := TEMP_AGENDA;
    for I in 1..NOP loop
      AP_FOUND := false;
      while SCHEDULE_INPUTS_LIST.NON_EMPTY(V(I)) and not AP_FOUND loop
        if SCHEDULE_INPUTS_LIST.VALUE(V(I)).THE_STOP > HBL then
          TEMP_LOWER := SCHEDULE_INPUTS_LIST.VALUE(V(I)).THE_LOWER;
          AP_FOUND := true;
          INSERT_IN_PRIORITY_QUEUE
                (SCHEDULE_INPUTS_LIST.VALUE(V(I)),TEMP_LOWER,QUE);
        else
          SCHEDULE_INPUTS_LIST.NEXT(V(I));
        end if;
      end loop;
    end loop;

    while PRIORITY_Q.NON_EMPTY(QUE) loop
      BEST := READ_BEST_FROM_PRIORITY_QUEUE(QUE);
      OP_NUM := BEST.THE_OPERATOR;
      for I in 1..NOP loop
        if SCHEDULE_INPUTS_LIST.NON_EMPTY(V(I)) and then
          SCHEDULE_INPUTS_LIST.VALUE(V(I)) = BEST then
          PROCESSOR_NUM := I;
          exit;
        end if;
      end loop;

      NEW_INPUT := BEST;
      TEMP := NEW_GRAPH.OP_RETURN(OP_NUM);
      OLD_LOWER := NEW_INPUT.THE_LOWER;
      DETERMINE_NEW_LOWER(TEMP,TEMP_AGENDA,NEW_INPUT);
      if NEW_INPUT.THE_LOWER = OLD_LOWER and then
              NEW_INPUT.THE_START = NEW_INPUT.THE_LOWER then
        REARRANGE_P := true;
```

```
        exit;
      else
        NEW_INPUT.THE_START := NEW_INPUT.THE_LOWER;
        NEW_INPUT.THE_STOP := NEW_INPUT.THE_START + TEMP.THE_MET;
        ADJUST_IT(TEMP,REARRANGE_P,NEW_INPUT,TEMP_AGENDA);
        if REARRANGE_P then
          exit;
        else
          SCHEDULE_INPUTS_LIST.NEXT(V(PROCESSOR_NUM));
          SCHEDULE_INPUTS_LIST.REMOVE(BEST,TEMP_AGENDA(PROCESSOR_NUM));
          REMOVE_BEST_FROM_PRIORITY_QUEUE(QUE);
          if SCHEDULE_INPUTS_LIST.NON_EMPTY(V(PROCESSOR_NUM)) then
            TEMP_LOWER:=
              SCHEDULE_INPUTS_LIST.VALUE(V(PROCESSOR_NUM)).THE_LOWER
            INSERT_IN_PRIORITY_QUEUE
            (SCHEDULE_INPUTS_LIST.VALUE(V(PROCESSOR_NUM)),TEMP_LOWER,QUE);
          end if;
        end if;
      end if;
    end loop;
  end if;

  if REARRANGE_P then
    ADJUST_PRECEDENCE(COUNT,PREC_LIST);
    NODE_LIST.DUPLICATE(PREC_LIST,P_LIST);
    SCHEDULE_1st_INSTANCES(HBL,READY,P_LIST,PROCESSOR_STOP,REV_AGENDA,ADDL_LIST);
    SCHEDULE_INPUTS_LIST.LIST_REVERSE(ADDL_LIST,A_LIST);
    SCHEDULE_INPUTS_LIST.FREE_LIST(ADDL_LIST);
    SCHEDULE_REST_OF_BLOCK(HBL,P_LIST,A_LIST,REV_AGENDA,PROCESSOR_STOP,INST_NO,READY);
    for I in 1..NOP loop
      SCHEDULE_INPUTS_LIST.FREE_LIST(TEMP_AGENDA(I));
      SCHEDULE_INPUTS_LIST.LIST_REVERSE(REV_AGENDA(I),TEMP_AGENDA(I));
    end loop;
  end if;
  TEST_SCHEDULE(HBL,TEMP_AGENDA,TEMP_COST);
  if TEMP_COST < BEST_COST then
    BEST_COST := TEMP_COST;
    for I in 1..NOP loop
      SCHEDULE_INPUTS_LIST.DUPLICATE(TEMP_AGENDA(I),BEST_AGENDA(I));
    end loop;
  end if;
```

```
        if TEMP_COST <= 0 then
          SOLUTION := true;
        elsif REARRANGE_P or else TEMP_COST <= PENALTY_COST or else
            RANDOM_NEXT < ANNEAL_FUNCTION(TEMP_COST,PENALTY_COST,TEMPERATURE) then
          ACCEPT_COUNT   :=   ACCEPT_COUNT + 1;
          PENALTY_COST   :=   TEMP_COST;
          for I in 1..NOP loop
            SCHEDULE_INPUTS_LIST.DUPLICATE(TEMP_AGENDA(I),AGENDA(I));
          end loop;
        else
          for I in 1..NOP loop
            SCHEDULE_INPUTS_LIST.DUPLICATE(AGENDA(I), TEMP_AGENDA(I));
          end loop;
        end if;
        TRIAL_COUNT := TRIAL_COUNT + 1;
      end loop;
      TEMPERATURE := TEMPERATURE * COOLING_FACTOR;
    end loop;
    AGENDA := BEST_AGENDA;
    PENALTY_COST := BEST_COST;

  exception

    when MATH.RANGE_ERROR =>
      TEXT_IO.PUT_LINE("THE FOLLOWING VALUES CAUSED A RANGE ERROR");
      TEXT_IO.PUT("PENALTY COST: ");
      TEXT_IO.SET_COL(15);
      int_io.put(PENALTY_COST, width=>5);
      TEXT_IO.NEW_LINE;
      TEXT_IO.PUT("TEMP COST: ");
      TEXT_IO.SET_COL(15);
      int_io.put(TEMP_COST, width=>5);
      TEXT_IO.NEW_LINE;
      TEXT_IO.PUT("TEMPERATURE: ");
      TEXT_IO.SET_COL(15);
      float_io.PUT(TEMPERATURE, fore=>5, aft=>5, exp=>0);
      TEXT_IO.NEW_LINE;
      AGENDA := (others=> null);

end ANNEAL_PROCESS;
```

```
  begin -- main SIMULATED ANNEAL

    NODE_LIST.DUPLICATE(PRECEDENCE_LIST,P_LIST);
    SCHEDULE_1st_INSTANCES(HBL,READY,P_LIST,PROCESSOR_STOP,REV_AGENDA,ADDL_LIST);
    SCHEDULE_INPUTS_LIST.LIST_REVERSE(ADDL_LIST,A_LIST);
    SCHEDULE_INPUTS_LIST.FREE_LIST(ADDL_LIST);
    SCHEDULE_REST_OF_BLOCK
        (HBL,P_LIST,A_LIST,REV_AGENDA,PROCESSOR_STOP,INSTANCE_NO,READY);
    for I in 1..NOP loop
      SCHEDULE_INPUTS_LIST.LIST_REVERSE(REV_AGENDA(I),AGENDA(I));
      SCHEDULE_INPUTS_LIST.FREE_LIST(REV_AGENDA(I));
    end loop;
    TEST_SCHEDULE(HBL,AGENDA,COST);

    if COST > 0 then
      VALID_SCHEDULE := false;
      RANDOM_INITIALIZE(2*COUNT + 1);
        --* Initialize Random Number Generator with an odd number.
      ANNEAL_PROCESS
          (HBL,AGEDA,VALID_SCHEDULE,COST,INSTANCE_NO,PRECEDENCE_LIST);
    else
      VALID_SCHEDULE := true;
    end if;

  end SIMULATED_ANNEAL;


end NEW_SCHEDULER_PKG;
```

LIST OF REFERENCES

[BAS74] Baker, K. R. and Su, Z. S., *"Sequencing with Due-Dates and Early Start Times to Minimize Maximum Tardiness"*, Naval Research Logistics Quarterly, 21, 1974.

[BAT83] Bannister, J. A. and Trivedi, K. S., *"Task Allocation in Fault-Tolerant Distributed Systems"*. In Acta Informatica, Springer-Verlag, 1983.

[BDW86] Blazewicz, J., Drabowski, M. and Weglarz, J., *"Scheduling Multiprocessor Tasks to Minimize Schedule Length"*. IEEE Trans., on Computer, C-35(5), 1986.

[BFR71] Bratley, P., Florian, M. and Robillard, P., *"Scheduling with Earliest Start and Due Date Constraints"*, Naval Research Logistics Quarterly, 18(4), Dec. 1971.

[BFR75] Bratley, P., Florian, M. and Robillard, P., *"Scheduling with Earliest Start and Due Date Constraints on Multiple Machines"*, Naval Research Logistics Quarterly, 22, 1975.

[BOE89] Borison, E., *"Program Changers and Cost of Selective Recompilation"*, Tech. Report CMU-CS-89-205, Computer Science Dept. Carnegie-Mellon University, Pittsburgh, 1989.

[BUR91] Burns, A., *"Scheduling Hard Real-Time Systems: A Review"*, Software Eng., Journal, 6, pp. 116-128, 1991.

[CER89] Cervantes, Julian J., *"An Optimal Static Scheduling Algorithm for Hard Real-Time Systems"*, M.S. thesis, NPS, Dec. 1989.

[DAD86] Davari, S. and Dhall, S. K., *"An On-Line Algorithm for Real-Time Tasks Allocation"*. In IEEE Real-Time Systems Symposium, Dec. 1986.

[DHL78] Dhall, S. K. and Liu, C. L., *"On a Real-Time Problem"*, Operations Research, 26(1), 1978.

[EFM83] Erschler, J., Fontan, G., Merce, C. and Roubellat, F., *"A New Dominance Concept in Scheduling N Jobs on a Single Machine with Ready Times and Due Dates"*, Operations Research, 31(1), 1983.

[ELS82] Elsayed, E. A., *"Algorithms for project scheduling with resource constraints"*, International journal of Production Research, 20(1), 1982.

[FAN90] Fan, Bao Hua, *"Evaluations of Some Scheduling Algorithms for Hard Real-Time Systems"*, M.S. thesis, NPS, Jun. 1990.

[FLO84] Flannery, B.P. and other, "*Numerical Recipes in C, The Art of Scientific Computing*", pp. 343-352, Cambridge University Press, 1984.

[HOR74] Horn, W.A., "*Some Simple Scheduling Algorithms*", Naval Research, 9, 1974.

[HUS90] Hsu, Liang Chuan, "*An Efficient Heuristic Scheduler for Hard Real-Time System*", M.S. thesis, NPS, Jun. 1990.

[JAN88] Janson, D.M., "*A Static Scheduler for the Computer Aided Prototyping System*", M.S. Thesis, Computer Science, NPS, Sep. 1988.

[JOH89] Johnson, D. S. and others, "*Optimization by Simulated Annealing: An Experimental Evaluation, Part I (Graph Portioning)*", Operation Research, v. 37 pp. 865-892, Nov.-Dec. 1989.

[KAN84] Kasahara, H. and Narita, S., *"Practical Multiprocessor Scheduling Algorithm for Efficient Parallel Processing"*, IEEE Transactions on Computer, Vol. c-33, No.11, pp.1023-1029, Nov. 1984.

[KIM89] Killic, Murat, "*Static Schedulers for Embedded Real-Time System*", MS. thesis, NPS, Dec. 1989.

[KIS78] Kise, H., "*A Solvable Case of the One-Machine Scheduling Problem with Ready and Due Times*", Operations Research, 26(1), 1978.

[LAF76] Lang, T. and Fernandez, E. B., *"Scheduling of Unit-Length Independent Tasks with Execution Constraints"*, Information Processing Letters, 4(4), 1976.

[LES86] Lehoczky, J. P. and Sha, L., "*Performance of Bus Scheduling Algorithm*". In Performance 86, 1986.

[LEV91] Levine, John Glenn, "*An Efficient Heuristic Scheduler for Hard Real-Time Systems*", M.S. thesis, NPS, Sep. 1991.

[LIL73] Liu, C. L. and Layland, J., "*Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*", J. ACM, 20(1), 1973.

[LUK88] Luqi and Ketabchi, M., "*A Computer Aided Prototype System*", Technical Report, NPS52-87-011,NPS. 1987 and in [IEEE Software, pp. 66-72], Mar. 1988.

[LUQ86] Luqi, *"Rapid Prototyping for Large Software System Design"*, Ph.D. Thesis, University of Minnesota, Duluth, Minnesota, May. 1986.

[LUQ89] Luqi, *"Software Evolution Through Rapid Prototyping"*, pp. 13-25, May. 1989.

[MAR88] Marlowe, Laura C., *"A Static Scheduler for Critical Timing Constraints"*, M.S. thesis, NPS, Dec. 1988.

[MOO68] Moore, J. M., *"An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs"*, Management Science, 15(1), 1968.

[OHE88] O'hern, J.T., "*A Conceptual Level Design for a Static Scheduler for Hard Real-Time System*", M.S. Thesis, Computer Science, NPS, Sep. 1988.

[OTV89] Otten, R. H. and Van Ginneken, L. P., "*The Annealing Algorithm*", Kluwer Academic Publishers, 1989.

[SIM80] Simons, B., "*A Fast Algorithm for Multiprocessor Scheduling*", Ln Proc. 21st Annual Symposium on Foundation of Computer Science, 1980.

[SIM83] Simons, B., *"Multiprocessor Scheduling of Unit-Time Jobs with Arbitrary Release Times and Deadlines"*, SIAM Journal for Computing, Dec. 1983.

[SIS84] Simons, B. and Sipser, M., "*On Scheduling Unit-Length Jobs with Multiple Release Time/Deadline Intervals*", Operations research, 32(1), 1984.

[STR88] Stankovic J. A., Ramamritham, K., "*Hard Real-Time System*", IEEE Computer Society Press, Washington, DC., 1988.

[TEI78] Teixeira, T., "*Static Priority Interrupt Scheduling*". In Proc. of the Seventh Texas Conference on Computing Systems, Nov. 1978.

[ULL75] Ullman, J.D., *"Np-Complete Scheduling Problem"*, Journal of Computer and System Sciences,Oct. 1975.

[ULL76] Ullman, J. D., "*Complexity of Sequence Problem*". In E. G. Coffman, editor, Computer and Job- Shop Scheduling Theory, J. Wiley New York, 1976.

[WHI89] White, J.L., "*The Development of a Rapid Prototyping Environment*", M.S. Thesis, Computer Science, NPS, Dec. 1989.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center — 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library — 2
   Code 52
   Naval Postgraduate School
   Monterey, CA 93943

3. Computer Science Department — 2
   Code CS
   Naval Postgraduate School
   Monterey, CA 93943

4. Office of the Assistant Secretary of the Navy — 1
   Research Development and Acquisition
   Department of the Navy
   Attn: Mr. Gerald A. Cann
   Washington, DC 20380-1000

5. Office of the Chief of Naval Operations — 1
   OP-094
   Department of the Navy
   Attn: VADM J. O. Tuttle, USN
   Washington, DC 20301-3040

6. Director of Defense Information — 1
   Office of the Assistant Secretary of Defense
   (Command, Control, Communications, & Intelligence)
   Attn: Mr. Paul Strassmann
   Washington, DC 20301-0208

7. Center for Naval Analysis — 1
   4401 Ford Avenue
   Alexandria, VA 22302-0268

8. Prof. Man-Tak Shing, Code CS/Sh — 5
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

9. Chairman, Code CS 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

10. Prof. Luqi, Code CS/Lq 10
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

11. Chief of Naval Research 1
Attn: ADM. Miller
800 N. Quincy Street
Arlington, VA 22217

12. Director, Ada Joint Program Office 1
OUSDRE (R&AT)
Room 3E114, The Pentagon
Attn: Dr. John P. Solomond
Washington, DC 20301-0208

13. Carnegie Mellon University 1
Software Engineering Institute
Attn: Dr. Dan Berry
Pittsburgh, PA 15260

14. Office of Naval Technology (ONT) 1
Code 227
Attn: Dr. Elizabeth Wald
800 N. Quincy St.
Arlington, VA 22217-5000

15. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn: Dr. B. Boehm
1400 Wilson Boulevard
Arlington, VA 22209-2308

16. Defense Advanced Research Projects Agency (DARPA) 1
ISTO
1400 Wilson Boulevard
Attn: LCol Eric Mattala
Arlington, VA 2209-2308

17. Defense Advanced Research Projects Agency (DARPA) 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

18. National Science Foundation 1
Division of Computer and Computation Research
Attn: K. C. Tai
Washington, DC 20550

19. Commander Space and Naval Warfare Systems Command 1
SPAWAR 3212
Department of the Navy
Attn: Cdr M. Romeo
Washington, DC 20363-5100

20. Office of Naval Research 1
Computer Science Division, Code 1133
Attn: Dr. Gary Koob
800 N. Quincy Street
Arlington, VA 22217-5000

21. Office of Naval Research 1
Computer Science Division, Code 1133
Attn: Dr. A. M. Van Tilborg
800 N. Quincy Street
Arlington, VA 22217-5000

22. Office of Naval Research 1
Computer Science Division, Code 1133
Attn: Dr. R. Wachter
800 N. Quincy Street
Arlington, VA 22217-5000

23. University of CA at Berkeley 1
Department of Electrical Engineering and
Computer Science
Computer Science Division
Attn: Dr. C.V. Ramamoorthy
Berkeley, CA 90024

24. University of MD 1
College of Business Management
Tydings Hall, Room 0137
Attn: Dr. Alan Hevner
College Park, MD 20742

25. University of MD 1
Computer Science Department
Attn: Dr. N. Roussapoulos
College Park, MD 20742

26. University of Massachusetts 1
Department of Computer and Information Science
Attn: Dr. John A. Stankovic
Amherst, MA 01003

27. University of Pittsburgh 1
Department of Computer Science
Attn: Dr. Alfs Berztiss
Pittsburgh, PA 15260

28. Commander, Naval Surface Warfare Center, 1
Code U-33
Attn: Dr. Philip Hwang
10901 New Hampshire Avenue
Silver Spring, MD 20903-5000

29. Attn: Joel Trimble 1
1211 South Fern Street, C107
Arlington, VA 22202

30. United States Laboratory Command 1
Army Research Office
Attn: Dr. David Hislop
P. O. Box 12211
Research Triangle Park, NC 27709-2211

31. Persistent Data Systems 1
75 W. Chapel Ridge Road
Attn: Dr. John Nester
Pittsburgh, PA 15238

32. Prof. Amr Zaky, Code CS/Za 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

33. Library of 1
Chung Shan Institute of Science and Technology
Lung-Tan, Tao-Yuan
Taiwan, R.O.C.

34. Library of 1
Chung Cheng Institute of Technology
Ta-Shi, Tao-Yuan
Taiwan, R.O.C.

35. Department of Electrical Engineering 1
Chung Cheng Institute of Technology
Ta-Shi, Tao-Yuan
Taiwan, R.O.C.

36. Department of Computer Science 1
Chung Cheng Institute of Technology
Ta-Shi, Tao-Yuan
Taiwan, R.O.C.

37. Computer Center 1
Chung Cheng Institute of Technology
Ta-Shi, Tao-Yuan
Taiwan, R.O.C.

38. Tzu-Chiang Chang 2
4 F, No.12, Alley 21, Lane 136,
Ming Chih Rd, Sec.2,
Tai-Shan, 243, Taipei,
Taiwan, R.O.C.

Thesis
C37185 Chang
c.1 Static scheduler for
hard real-time tasks on
multiprocessor systems.